Lecture 6.1

Transport Layer: Process-to-Process Delivery: UDP and TCP

Dr. Vandana Kushwaha

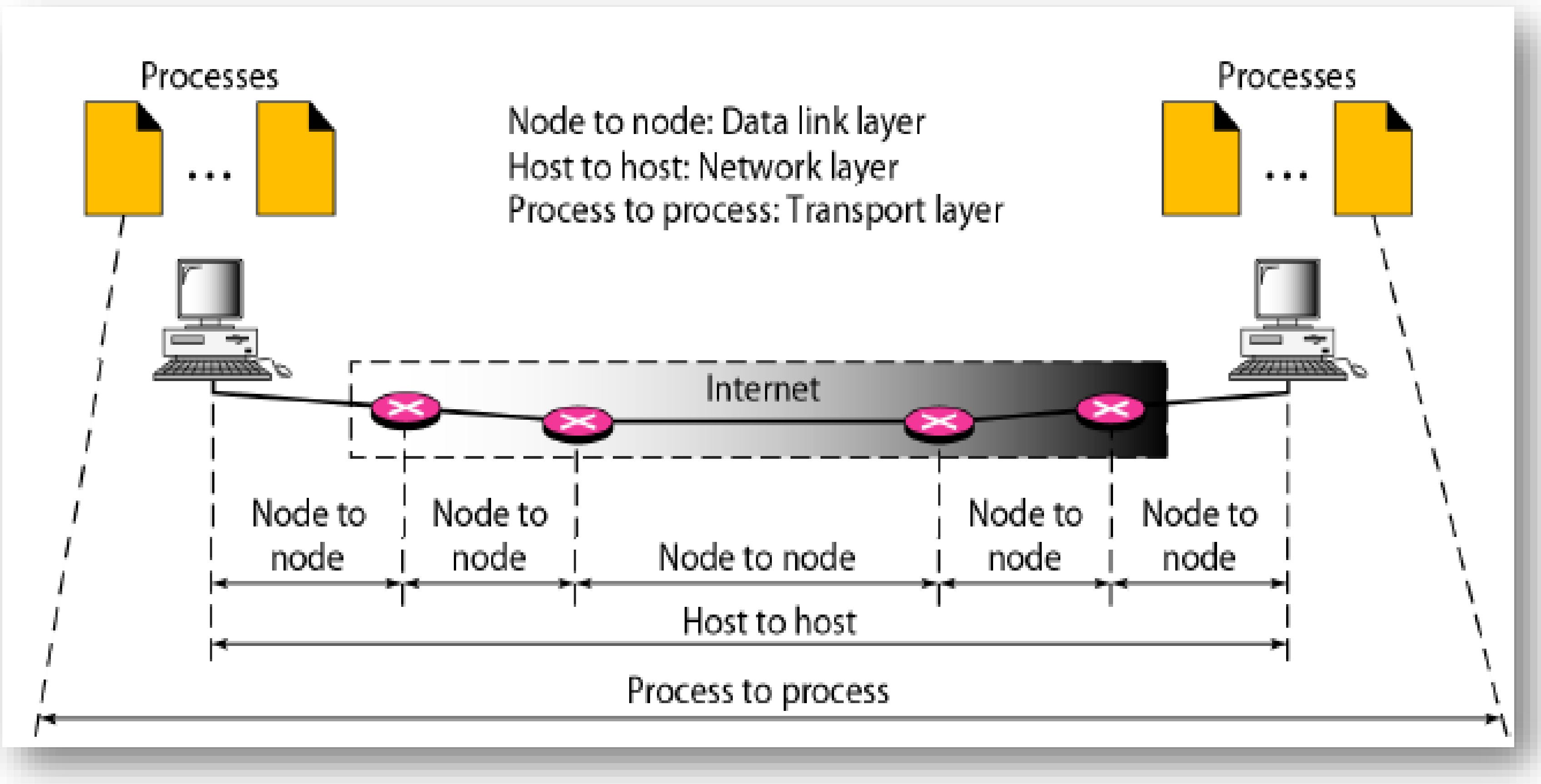
Department of Computer Science

Institute of Science, BHU, Varanasi

PROCESS-TO-PROCESS DELIVERY

- **Communication** on the **Internet** is not defined as the **exchange** of **data** between **two nodes** or between **two hosts.**
- But **real communication** takes place **between two processes** (**application programs**), ie. we need **process-to-process delivery**.
- However, at any moment, **several processes** may be **running** on the **source host** and **several** on the **destination host.**
- To **complete** the **delivery,** we need a **mechanism** to **deliver data** from **one** of these **processes** running on the **source host** to the **corresponding process** running on the **destination host.**
- The **Transport layer** is **responsible** for **process-to-process delivery**-the delivery of a packet, part of a message, from one process to another.

PROCESS-TO-PROCESS DELIVERY



Client/Server Paradigm

- Although there are **several ways** to achieve **process-to-process** **communication**, the **most common** one 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.**
- **Both processes** (**client** and **server**) have the **same name**.
- For **example**, to open a **web page** from a **remote machine(Server)**, we need a **HTTP client** **process** running on the **local host(Client)** and a **HTTP server** **process** running on a **remote machine(Server).**
- A **remote computer** can run **several server programs** at the **same time**, just as **local computers** can run one or more **client programs** at the **same time.**

Port Address

- In the **Internet model**, the **port numbers** are **16-bit integers** between **0** and **65,535**.
- The **client program** defines **itself** with a **port number,** chosen **randomly** by the **transport layer software** running on the **client host.**
- This is the **ephemeral** **port number**.
- The **server process** must also **define itself** with a **port number**.
- This **port number**, however, **cannot** **be chosen randomly**.
- The **Internet** has **decided** to use **universal port numbers** for **servers**; these are called **well-known** **port numbers**.

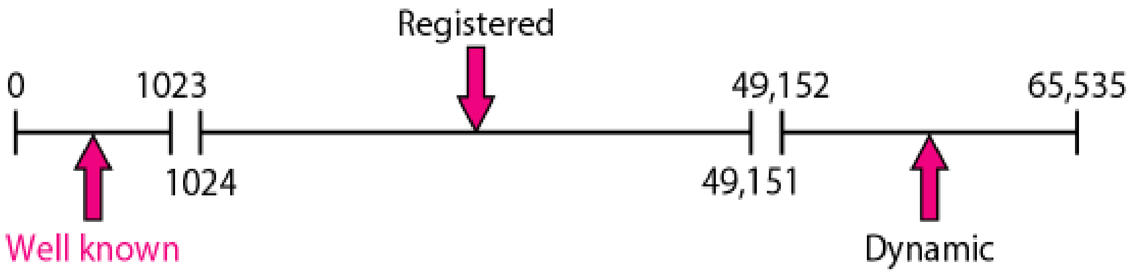
Types of Port Numbers

- The **IANA (Internet Assigned Number Authority)** has **divided** the **port numbers** into **three ranges:** *well known*, *registered*, and *dynamic (or private).*

i. **Well-known ports.** The ports ranging from **0 to 1023** are assigned and controlled by **IANA**. These are the well-known ports.

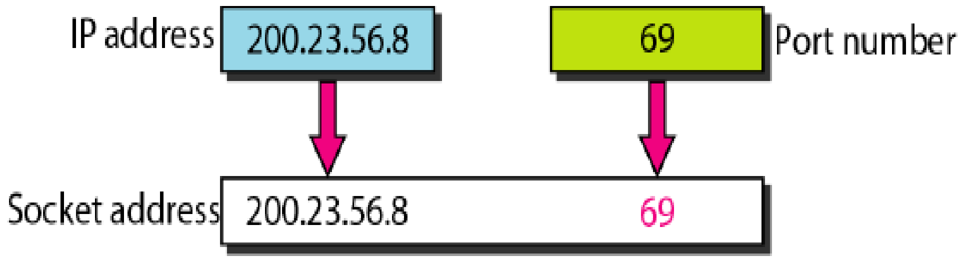
ii. **Registered ports.** The ports ranging from **1024 to 49,151** are not assigned or controlled by **IANA**. They can only be registered with **IANA** to prevent duplication. These are used by **vendors** for their **own server applications**.

iii. **Dynamic ports.** The ports ranging from **49,152 to 65,535** are neither controlled nor registered. They can be used by any **process**. These are the **ephemeral ports**.



Socket Addresses

- **Process-to-process delivery** needs **two identifiers**, **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**.



Connectionless Versus Connection-Oriented Service

- A **transport layer protocol** can either be **connectionles**s or **connection-oriented.**
- ***Connectionless Service***
- In a **connectionless service**, the **packets** are sent from one party to another with no need for connection establishment or connection release.
- The **packets** are **not numbered**; they may be **delayed** or **lost** or may arrive **out of sequence.**
- There is **no acknowledgment** either.
- ***Connection Oriented Service***
- In a **connection-oriented service**, a **connection** is **first established** between the **sender** and the **receiver**, **data** are **transferred.**
- At the **end**, the **connection** is **released.**

Reliable Versus Unreliable

- The **Transport layer service** can be **reliable** or **unreliable**.
- If the **application layer program** needs **reliability**, we use a **reliable transport layer protocol** by implementing **flow** and **error control** at the **transport layer**.
- This means a **slower** and more **complex service.**
- **TCP** is a **connection oriented** and **reliable delivery protocol**.
- On the other hand, if the **application program does not need reliability** because
  - It uses its **own flow** and **error control** mechanism or
  - It **needs fast service** or
  - The **nature** of the **service** does **not demand flow** and **error control** (real-time applications),
- Then an **unreliable protocol** can be **used**.
- **UDP** is **connectionless** and **unreliable delivery protocol**.

Protocols at Transport Layer

The original **TCP/IP protocol suite** specifies **two protocols** for the **transport layer**: **UDP and TCP.**

USER DATAGRAM PROTOCOL (UDP)

- The ***User Datagram Protocol (UDP)*** *is called a* ***connectionless, unreliable transport protocol*****.**
- It **does not add** anything to the **services** of **IP** except to provide **process-to process communication** instead of **host-to-host** communication.
- Also, it performs **very limited** error checking.
- **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** by using **UDP** takes much **less interaction** between the **sender** and **receiver** than using **TCP.**

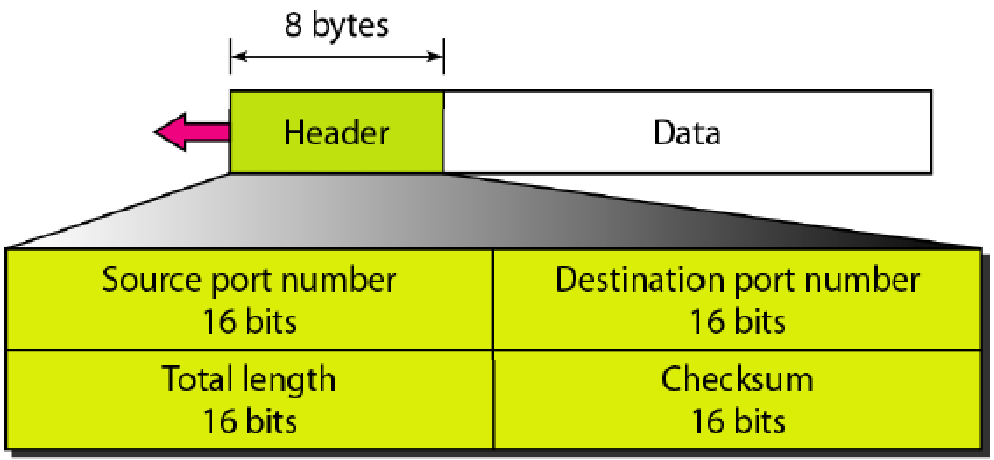
Format of User Datagram

UDP packets, called **user datagrams**, have **a fixed-size header of 8 bytes**. The **fields** are as follows:

i. Source port number.

- This is the **port number** used by the **process** running on the **source host.**
- It is **16 bits long**, which means that the **port number** can **range from 0 to 65,535**.
- If the **source host** is the **client** (a client sending a request), the **port number**, in most cases, is an **ephemeral port number** requested by the **process** and **chosen** by the **UDP software** running on the **source host.**
- If the **source host** is the **server** (a server sending a response), the **port number**, in most cases, is a **well-known port number**.

Format of User Datagram



ii. Destination port number.

- This is the **16 bits** long **port number** used by the **process** running on the **destination host.**
- If the **destination host** is the **server** (a client sending a request), the port number, in most cases, is a **well-known port number**.
- If the **destination host** is the **client** (a server sending a response), the port number, in most cases, is an **ephemeral port number**. In this case, the **server** copies the **ephemeral port number** it has received in **the request packet.**

Format of User Datagram

iii. Length.

- This is a **16-bit field** that 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 a **total length** of **65,535 bytes**.
- The **length field** in a **UDP user datagram** is actually **not necessary**.
- A **user datagram** is **encapsulated** in an **IP datagram.**
  - **UDP length = IP length - IP header's length**

iv. Checksum. This field is used to **detect errors** over the entire user datagram (header plus data).

UDP Operation

Connectionless Services

- As mentioned previously, **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**.
- Also, there is **no connection establishment** and **no connection termination**, as is the case for **TCP.**
- This means that **each user datagram** can **travel** on a **different path**.

UDP Operation

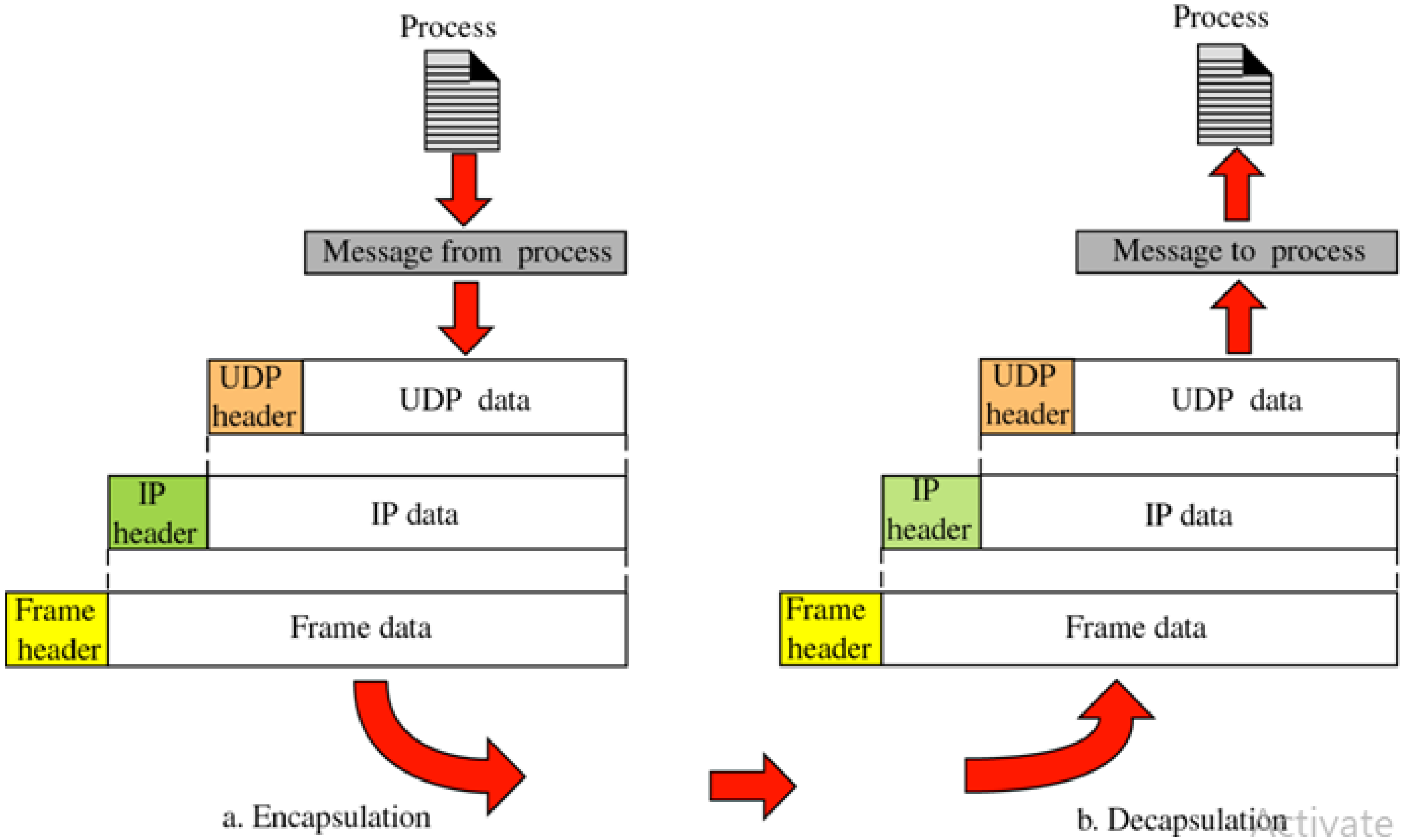
Flow and Error Control

- **UDP** is a **very simple**, **unreliable** transport protocol.
- There is **no flow control** and hence no **window mechanism.**
- The **receiver may overflow** with **incoming messages.**
- 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.**

Encapsulation and Decapsulation

- To **send** a **message** from one **process** to another, the **UDP** protocol **encapsulates** and **decapsulates** messages in an **IP datagram.**

UDP Encapsulation and Decapsulation Process



Use of UDP

- **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** **(File Transfer Protocol)** 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.**
- **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** **(Simple Network Management Protocol).**
- **UDP** is used for some **route updating protocols** such as **Routing Information Protocol (RIP).**

TCP (Transmission Control Protocol)

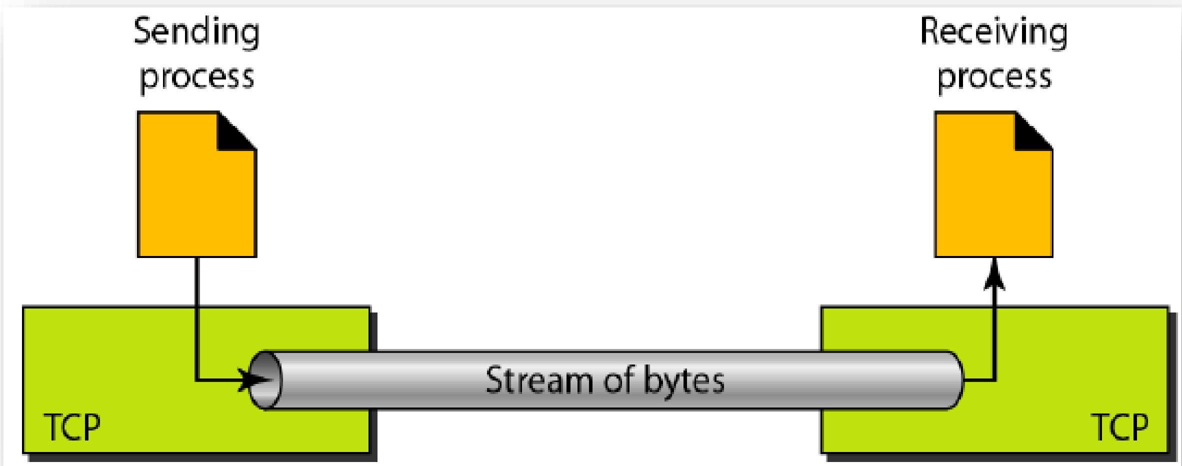
- **TCP (Transmission Control Protocol)** is a ***connection-oriented, reliable*** transport protocol.
- It adds **connection-oriented** and **reliability** features to the **services** of **IP**.
- In addition, **TCP** uses **flow** and **error control** mechanisms at the **transport level.**
- **Services offered by TCP**
- **i. Process-to-Process Communication**
- Like **UDP, TCP** provides **process-to-process communication** using **port numbers.**

TCP Services

ii. Stream Delivery Service

- **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 **data** across the **Internet.**
- The **sending process produces** (writes to) the **stream of bytes**, and the **receiving process consumes** (reads from) them.
- The **sending** and the **receiving processes** may not **write** or **read data** at the **same speed.**
- There are **two buffers**, the **sending buffer** and the **receiving buffe**r, one for each direction.

TCP Services: Stream Delivery Service



TCP Services

iii. Segments

- TCP **groups a number of bytes** together into a **packet** called a **segment**.
- TCP adds a **header** to each **segment** (for control purposes) and **delivers** the **segment** to the **IP layer** for transmission.
- The **segments** are **encapsulated in IP datagrams** and transmitted. The **segments** are **not necessarily the same size**.

iv. Full-Duplex Communication

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

Connection-Oriented & Reliable Service

- **TCP,** unlike **UDP**, is a **connection-oriented protocol.**
- When a **process** at **site A** wants to **send** and **receive data** from **another process** at **site B**, the following **three events** occur:
  1. The **two TCPs establish a connection** between them.
  2. Data are exchanged in both directions.
  3. The **connection is terminated.**
- **TCP** is a **reliable** transport protocol.
- It uses an **acknowledgment mechanism** to **check** the **safe and sound arrival** of **data.**

TCP Features

A. Numbering System

TCP software keeps track of the **segments** being **transmitted** or **received** using **two fields** called the **sequence number** and the **acknowledgment number**.

i. Byte Number

- **TCP numbers** all **data bytes** that are transmitted in a connection.
- Numbering is **independent** in **each direction.**
- When **TCP** receives bytes of data from a process, it stores them in the **sending buffer** and **numbers** them.
- The **numbering** does not necessarily start from **0.**
- Instead, **TCP generates** a **random number** between **0 and 2^{32} - 1** for the number of the **first byte.**
- For **example,** if the **random number** happens to be **1057** and the **total data** to be **sent** are **6000 bytes**, the **bytes** are **numbered** from **1057** to **7056.**

TCP Features

ii. Sequence Number

- After the **bytes** have been **numbered**, **TCP** assigns a **sequence number** to **each segment** that is being sent.
- *The **sequence number** for each segment is the **number of the first byte carried in that segment.***

Example : Suppose a TCP connection is transferring a file of 5000 bytes. The first byte is numbered 10,001. What are the sequence numbers for each segment if data are sent in five segments, each carrying 1000 bytes?

Solution

The following shows the sequence number for each segment:

- Segment 1 Sequence Number: 10,001 (range: 10,001 to 11,000)
- Segment 2 Sequence Number: 11,001 (range: 11,001 to 12,000)
- Segment 3 Sequence Number: 12,001 (range: 12,001 to 13,000)
- Segment 4 Sequence Number: 13,001 (range: 13,001 to 14,000)
- Segment 5 Sequence Number: 14,001 (range: 14,001 to 15,000)

TCP Features

iii. Acknowledgment Number

- The **acknowledgment number** defines the **number** of the **next byte** that the **receiver** **expects** to **receive.**
- In addition, the **acknowledgment number** is **cumulative**, which means that the receiver takes the number of the **last byte** that it has **received**, safe and sound, **adds 1** to it, and **announces** this **sum** as the **acknowledgment number**.
- The term ***cumulative*** here means that if a receiver uses **5643** as an **acknowledgment number**, it has **received** all **bytes** from the **beginning** up to **5642.**
- Note that this does not mean that the **receiver** has **received 5642 bytes** because the **first byte number** does not have to start from **0.**

TCP Features

B. Flow Control

- TCP, unlike UDP, provides ***flow control***.
- The **receiver** of the data controls the **amount of data** that are to be sent by the sender.
- This is done to prevent the receiver from being **overwhelmed with data**.
- The numbering system allows TCP to use a **byte-oriented flow control**.

C. Error Control

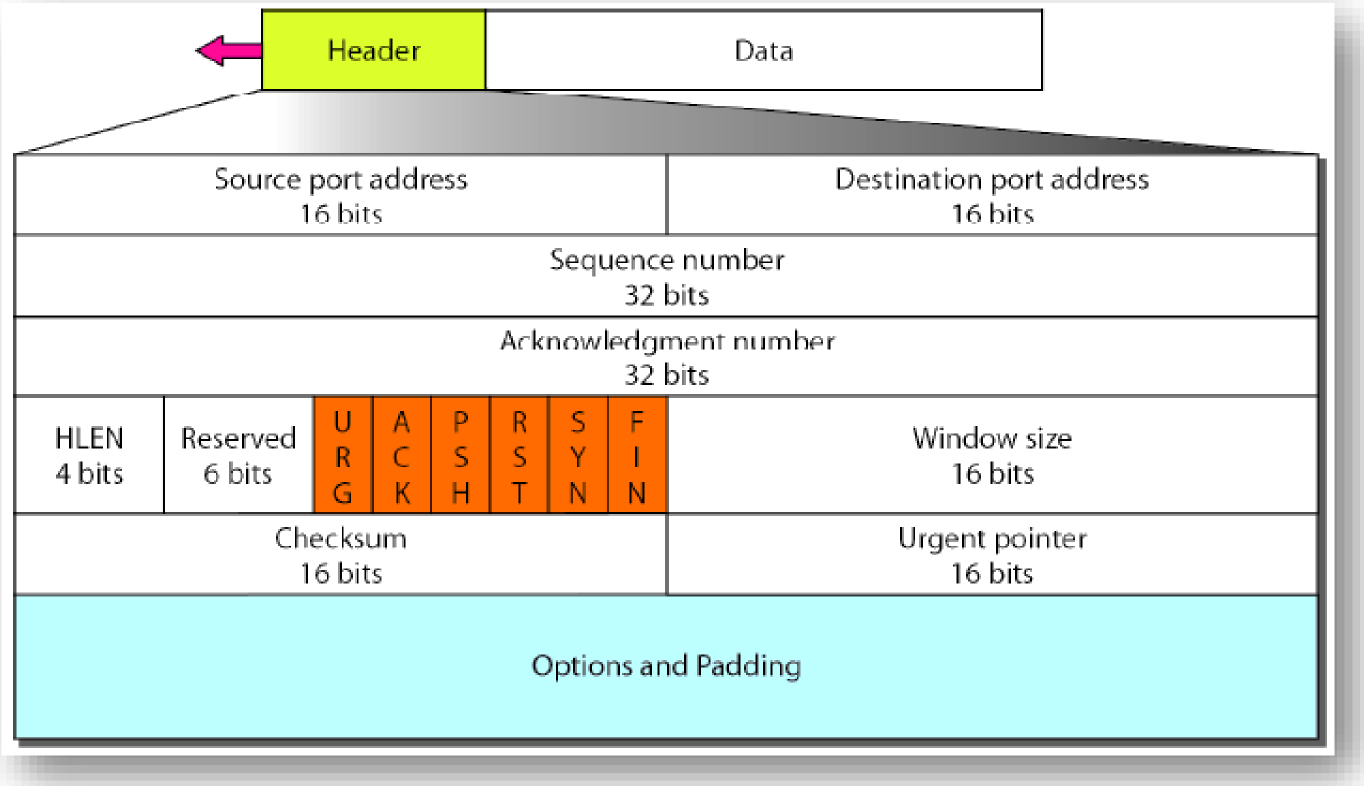
- To provide **reliable service**, **TCP** implements an **error control mechanism**.
- Although **error control** considers a **segment** as the **unit of data** for error detection (loss or corrupted segments), error control is **byte-oriented**.

D. Congestion Control

- **TCP**, unlike **UDP**, takes into account **congestion** in the **network.**
- The **amount of data** sent by a **sender** is not only controlled by the **receiver** (flow control), but is also **determined** by the **level of congestion** in the **network**.

TCP Segment Format

- The **TCP segment** consists of a **20** to **60 byte** **header**, 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.**



TCP Segment Format

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.**
- During **connection establishment**, each party uses a **random number generator** to create an **initial sequence number** (ISN), which is usually different in each direction.

TCP Segment Format

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 defines ***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** can be **between** **5** (5 x 4 =20) and **15** (15 x 4 =60).

Reserved.

- This is a **6-bit** field **reserved** for **future use.**

TCP Segment Format

Control Flags

- This field defines **6 different control bits** or **flags** as shown in below.
- **One** or **more** of these **bits** can be **set** at a time.

Flag	*Description*
URG	The value of the urgent pointer field is valid.
ACK	The value of the acknowledgment field is valid.
PSH	Push the data.
RST	Reset the connection.
SYN	Synchronize sequence numbers during connection.
FIN	Terminate the connection.

TCP Segment Format

Window size

- This **field** defines the **size of the window**, in **bytes**, that the **other party must maintain**.
- 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.

TCP Segment Format

Checksum

- This **16-bit field** contains the checksum for **header+data** **security**.

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** the **number** 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.**

A TCP Connection

- **TCP** is **connection-oriented** protocol**.**
- A **connection-oriented transport protocol** establishes a **virtual path between the source** and **destination.**
- All the **segments** belonging to a **message** are then **sent over** this **virtual path**.
- Using a **single virtual pathway** for the **entire message** facilitates the **acknowledgment process** as well as **retransmission** of **damaged** or **lost frames.**
- In TCP, **connection-oriented transmission** requires **three phases:**

    1. Connection establishment,

    2. Data transfer,

    3. Connection termination.

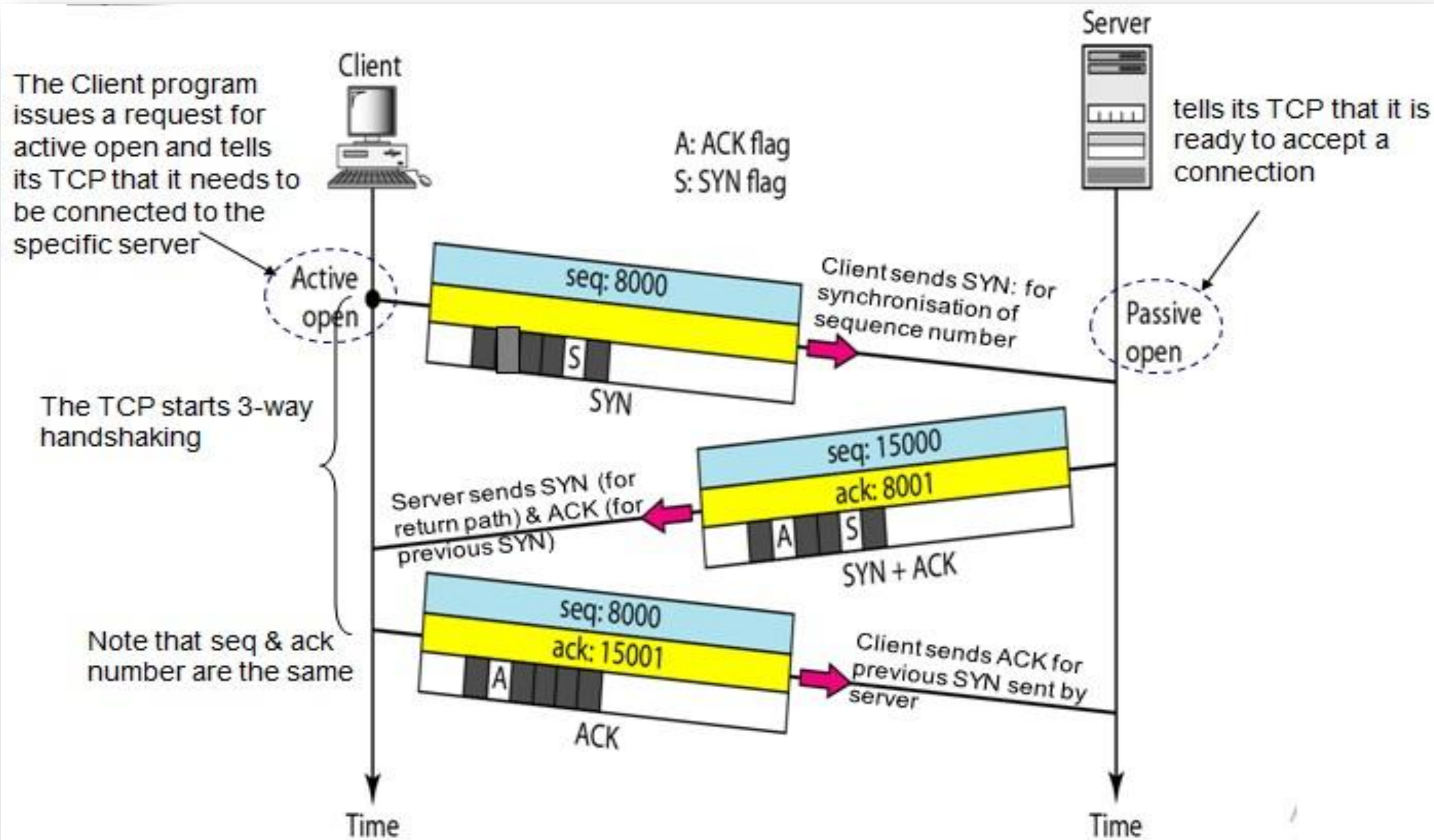
1. Connection Establishment

- **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.**
- 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.**

Connection Establishment

- This is called a **request** for a ***passive open.***
- Although the **server TCP** is **ready** to **accept** any **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** that it **needs** to be **connected** to that **particular server**.
- **TCP** can now **start** the **three-way handshaking process** as shown in **Figure** on next slide.
- To show the process, we use **two time lines**: one at each site.
- Each **segment** has **value** for all its **header fields.**

Connection Establishment using three-way handshaking



Connection establishment using three-way handshaking

- The **three steps** in this **phase** are as follows:

Step 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.**
- A **SYN segment consumes one sequence number**.
- When the **data transfer** starts, the **sequence number** is **incremented by 1**.
- Thus a **SYN segment cannot carry data**, but it **consumes one sequence number**.

Connection establishment using three-way handshaking

Step 2:

- The **Server** sends the **second segment,** a **SYN +ACK** segment, with **2 flag bits set**: **SYN** and **ACK**.
- This **segment** has a **dual purpose**.
- It is a **SYN segment** for **communication** in the other direction and serves as the **acknowledgment** for the **SYN segment.**
- It **consumes one sequence number**.
- **A SYN +ACK segment cannot carry data, but does consume one sequence number.**

Connection establishment using three-way handshaking

Step 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.**
- Note that the **sequence number** in this **segment** is the **same** as the one in the **SYN segment**;
- ***The ACK segment does not consume any sequence numbers.***
- ***An ACK segment, if carrying no data, consumes no sequence number.***

Data Transfer

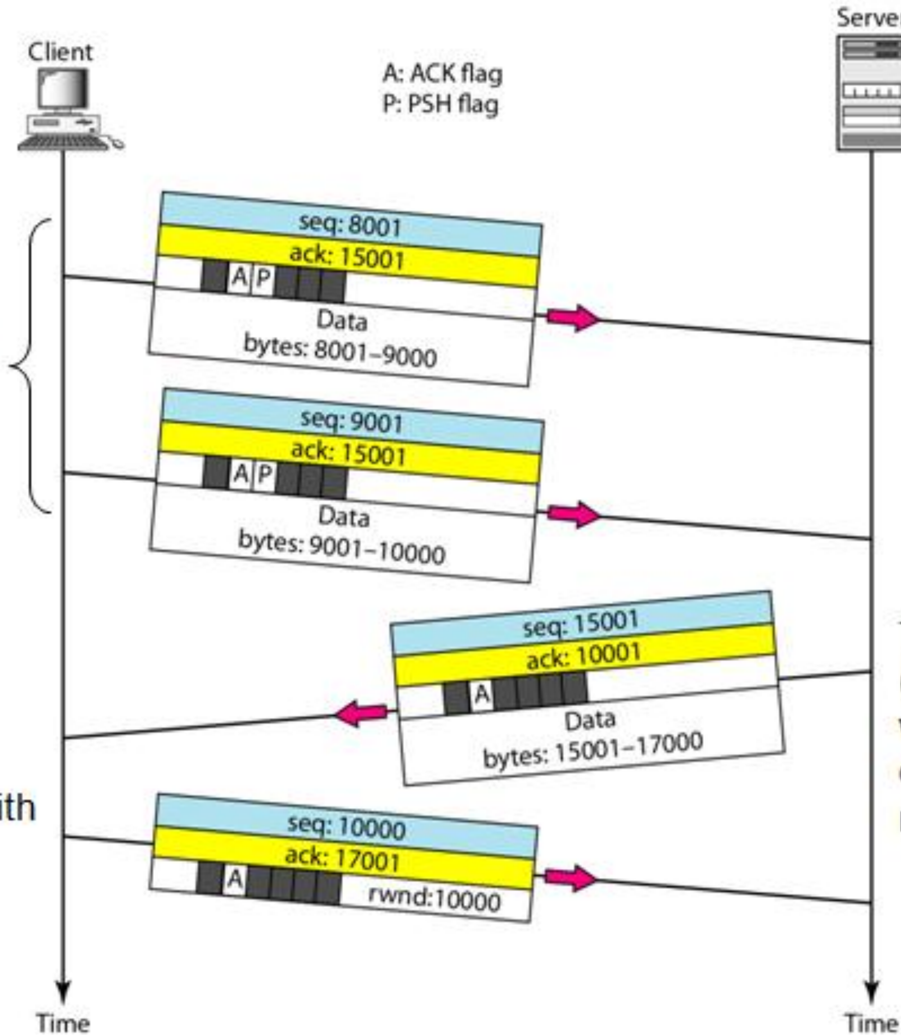
- After **connection is established**, **bidirectional data transfer** can take place.
- The **client** and **server** can **both send data** and **acknowledgments.**
- The **acknowledgment** is **piggybacked** with the **data.**
- **Figure** on next slide shows an **example. In** this **example,** after **connection** is **established** (not shown in the figure), the **client** sends **2000 bytes** of **data** in **two segments.**
- The **server** then sends **2000 bytes** in **one segment**.
- The **client** sends **one more** segment.
- The **first three segments** carry **both data** and **acknowledgment,** but the **last segment** carries **only** an **acknowledgment** because there are **no more data** to be sent.

Data Transfer

- Note the **values** of the **sequence** and **acknowledgment numbers.**
- The **data segments** sent by the **client** have the **PSH** (push) **flag set** so that the **server TCP** knows to **deliver data** to the **server process** as soon as they are **received.**
- The **segment** from the **server,** on the **other hand**, does **not set** the **push flag.**

Data Transfer

The Client sends two 2000 bytes of data in 2 segments at the same time acknowledged previous sending by server: 15001



The Server sends ACK (10001) and piggybacked with another 2000 bytes of data starting with sequence number 15001

The Client ACK with 17001

Pushing Data

- The **sending TCP uses a buffer** to **store** the **stream of data** coming from the **sending application program.**
- The **receiving TCP also buffers the data** when they **arrive** and delivers them to the **application program** when the **application program** is **ready.**
- This type of **flexibility** increases the **efficiency of TCP**.
- However, on occasion the **application program** has **no need** for this **flexibility.**
- For **example,** consider an **application program** that communicates **interactively** with another **application program** on the **other end.**
- The **application program** on **one site** wants to **send** a **keystroke** to the **application** at the **other site** and want to **receive** an **immediate response.**
- **Delayed transmission** and **delayed delivery** of **data** may **not be acceptable** by the **application program.**

Pushing Data

- **TCP** can **handle** such a **situation.**
- The **application program** at the **sending site** can **request a *push* operation**.
- This means that the **sending TCP** must **not wait for the window to be filled**.
- It must **create a segment** and **send it immediately**.
- The **sending TCP** must also **set** the **push bit (PSH)** to let the **receiving TCP** know that the **segment** includes **data** that **must be delivered** to the **receiving application** program **as soon as possible** and **not to wait** for **more data to come**.

Urgent Data

- At **some occasion** an **application program** needs to send ***urgent* bytes**.
- This means that the **sending application program** wants a **piece of data** to be **read out of order** by the **receiving application program.**
- As an **example,** suppose that the **sending application program** is **sending data** to be **processed** by the **receiving application program**.
- When the **result** of processing **comes back**, the **sending application program** finds that **everything is wrong.**
- It wants to **abort** the **process**, but it has **already sent** a **huge amount** of **data.**
- If it issues an **abort command** (**control +C**), these **two characters** will be **stored** at the **end** of the **receiving TCP buffer**.
- It will be **delivered** to the **receiving application program** after **all the data** have been **processed.**

Urgent Data

- The **solution** is to **send a segment** with the **URG bit set**.
- The **sending application program** tells the **sending TCP** that the **piece of data** is **urgent.**
- The **sending TCP** creates a **segment** and **inserts the urgent data at the beginning of the segment.**
- The **rest of the segment** can contain **normal data** from the buffer**.**
- **The urgent pointer field in the header defines the end of the urgent data and the start of normal data**.
- When the **receiving TCP** receives a **segment** with the **URG bit set**, it **extracts** the **urgent data** from the **segment**, using the **value** of the **urgent pointer**, and **delivers** them, **out of order**, to the **receiving application program.**

URG vs PSH

- **PSH**
  - **Send** this **message** to the **application** **right now** — **don’t wait**.
  - **Flushes** the **TCP buffer** and **pushes data** to the **application.**

- **URG**
  - This **part** of the **message** is **important** — **handle it first**.
  - May **bypass** **normal data** flow for **specific bytes.**

Connection Termination in TCP

- Any 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**:

i. ***Three-way handshaking*** for **full-close**

ii. ***Four-way handshaking*** for **half-close**

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

Connection Termination in TCP

Step 1

- In a **normal 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**.
- Note that a **FIN segment** can **include** the **last chunk of data** sent by the **client,** or it can be just a **control segment.**
- If it is **only** a **control segment**, it **consumes** only **one sequence number.**

Connection Termination

Step 2

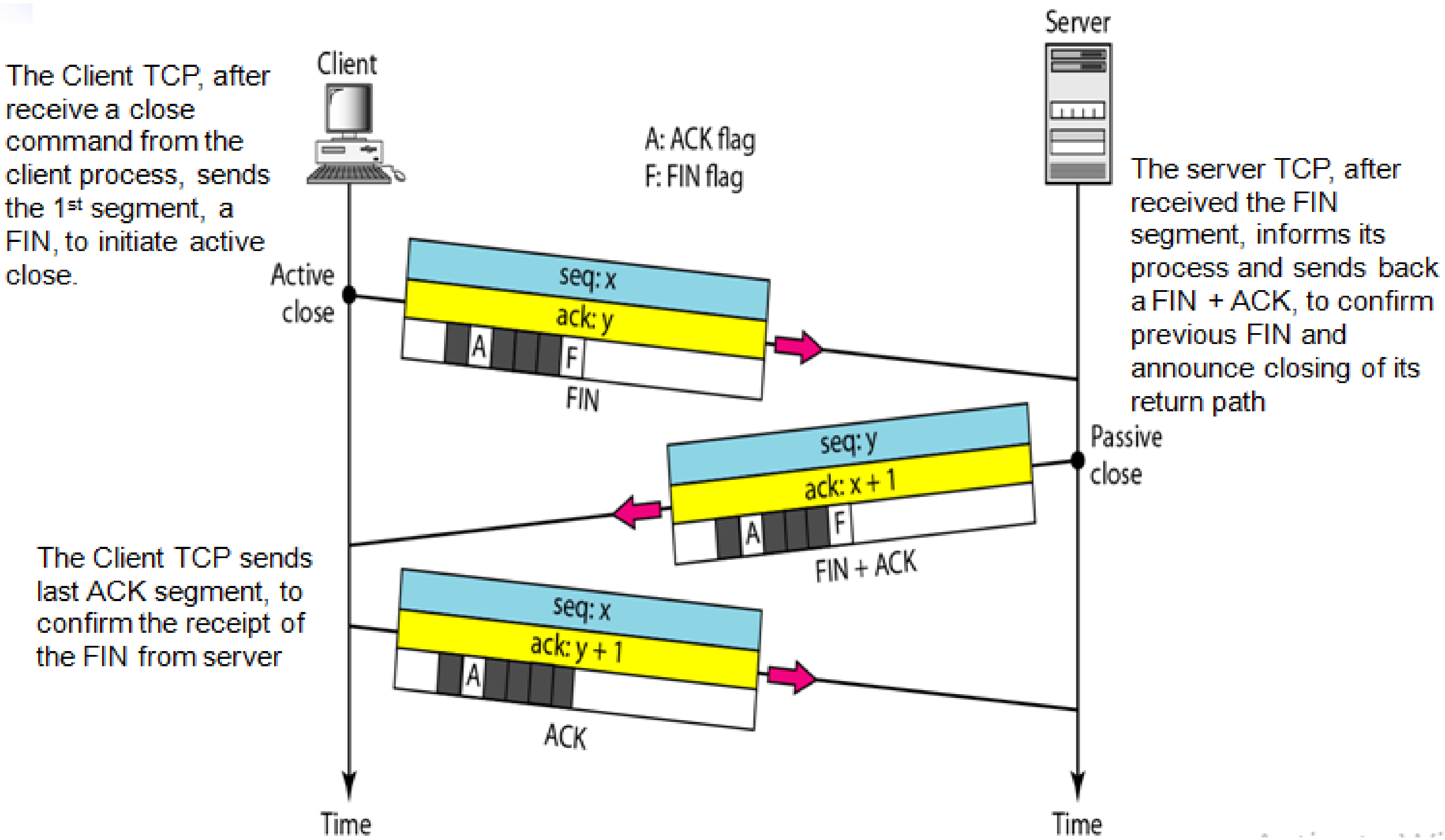
- The **server TCP**, after **receiving** the **FIN segment**, **informs** its **process** of the **situation** and **sends** the **second segment**, a **FIN +ACK segment,** to **confirm the receipt of the FIN segment** from the **client** and at the same time to **announce the closing of the connection** in the **other direction.**
- This **segment** can also **contain the last chunk of data** from **the server**.
- If it **does not carry data**, it **consumes** only **one sequence number.**

Connection Termination

Step 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 **1 plus** the **sequence number received** in the **FIN segment** from the **server.**
- This **segment cannot carry data** and **consumes no sequence numbers**.

Connection termination using three-way handshaking



Half-Close

- In **TCP**, ***one end can stop sending data*** *while* ***still receiving data***. This is called a **half-close**.
- Although **either end** can **issue** a **half-close,** it is **normally initiated by the client**.
- 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 the **data**, **can close the connection in the outbound direction.**
- However, **the inbound direction must remain open to receive the sorted data.**
- The **server,** after **receiving** the **data,** still **needs time** for **sorting;** its **outbound direction** must remain **open.**

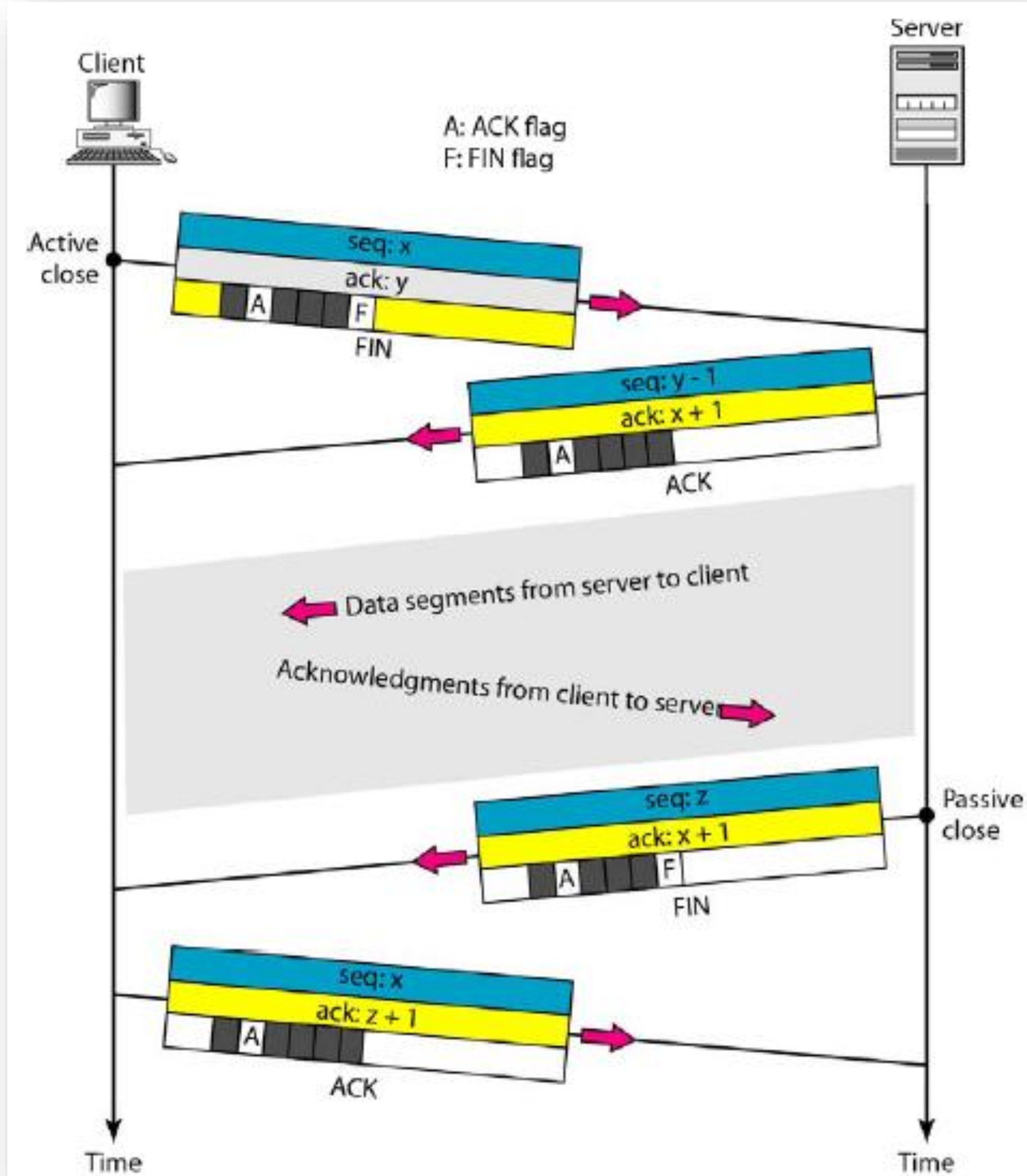
Half-Close

- The **client** **half-closes** the **connection** by **sending** a **FIN segment**.
- The **server** accepts the **half-close** by **sending** the **ACK segment**.
- The **data transfer** from the **client** to the **server** **stops.**
- The **server**, however, **can** **still** **send data.**
- When the **server** has **sent** **all** the **processed data**, it sends a **FIN segment**, which is **acknowledged** by an **ACK** from the **client.**
- After **half-closing** of the connection, **data** **can** **travel** **from the** **server** **to the** **client** and **acknowledgments** **can** **travel** **from the** **client** **to the** **server**.

Half-Close

- The **client cannot send** any **more data** to the **server.**
- Although the **client** has **received sequence number *y* - 1** and is **expecting *y*,** the **server sequence number** is still ***y* - 1.**
- When the **connection** finally **closes,** the **sequence number** of the **last ACK segment** is **still *x*,** because **no sequence numbers** are **consumed** during **data transfer** in that **direction.**

Half-Close

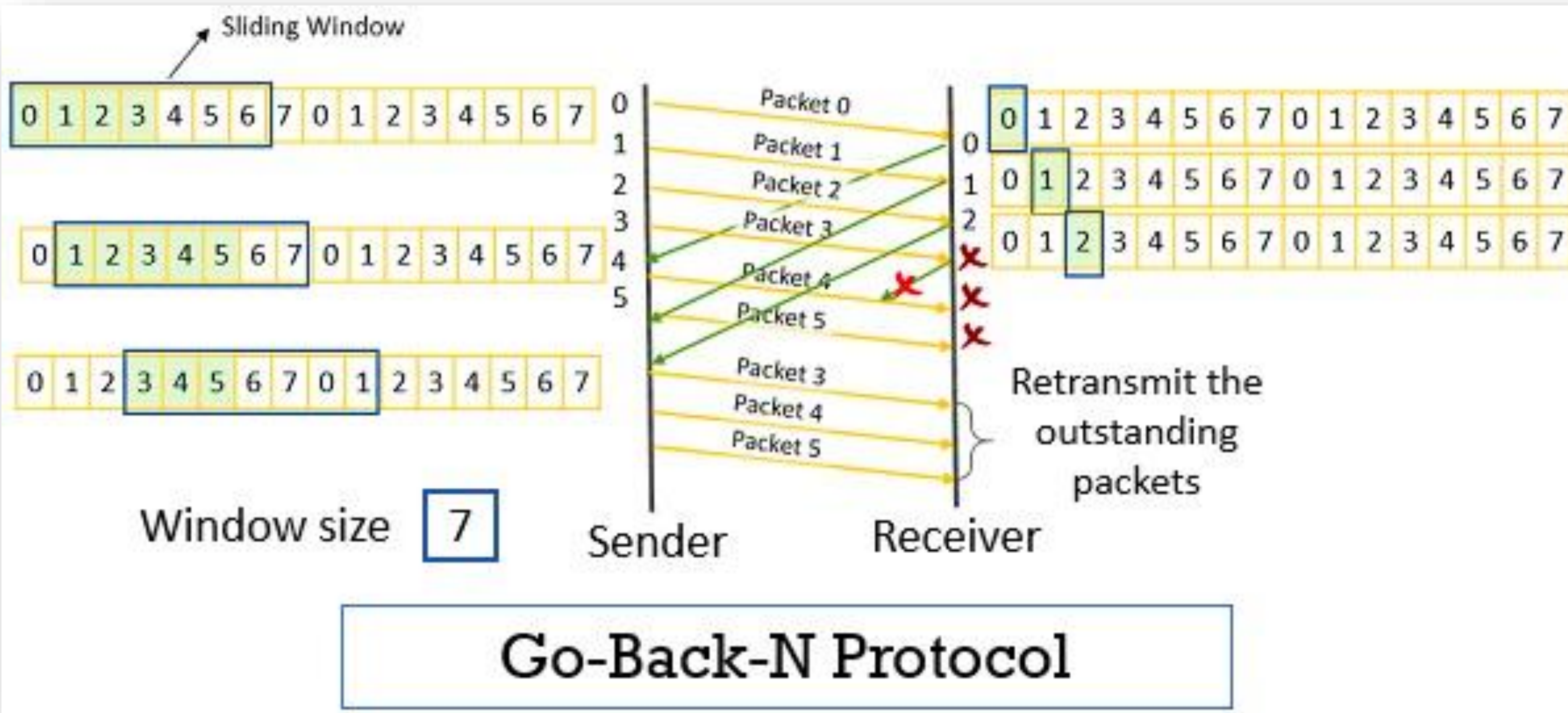


Client
Server
A: ACK flag
F: FIN flag
Active close
seq: x
ack: y
A
F
FIN
seq: y - 1
ack: x + 1
A
ACK
Data segments from server to client
Acknowledgments from client to server
Passive close
seq: z
ack: x + 1
A
F
FIN
seq: x
ack: z + 1
A
ACK
Time
Time

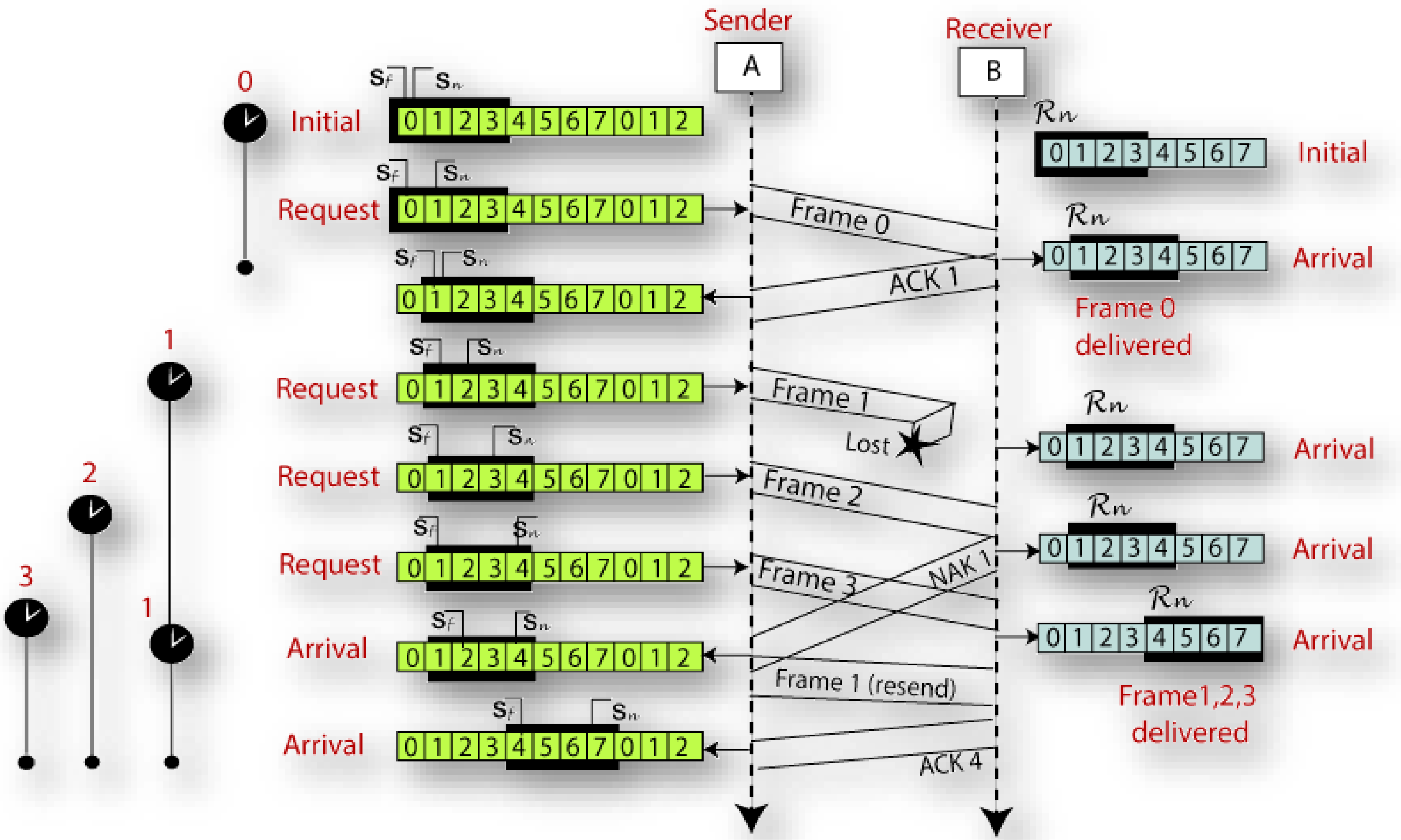
Flow Control

- **TCP** uses a **sliding window mechanism,** to handle **flow control.**
- The **Sliding window protocol** used by **TCP,** however, is something **between** the ***Go-Back-N*** and **Selective Repeat sliding window mechanisms.**
- The **sliding window protocol** in **TCP** looks like the **Go-Back-N protocol** because it **does not use NAKs**; it looks like **Selective Repeat** because the **receiver holds the out-of-order segments** until the **missing ones arrive.**
- There are **two big differences** between this **sliding window** and the one we used at the **data link layer.**
- First, the **sliding window of TCP is byte-oriented**; the **sliding window** in the **data link layer** is **frame-oriented**.
- Second, the **TCP's sliding window is of variable size**; the **sliding window** in the **data link layer** was of **fixed size**.

Go-Back-N at Data Link Layer

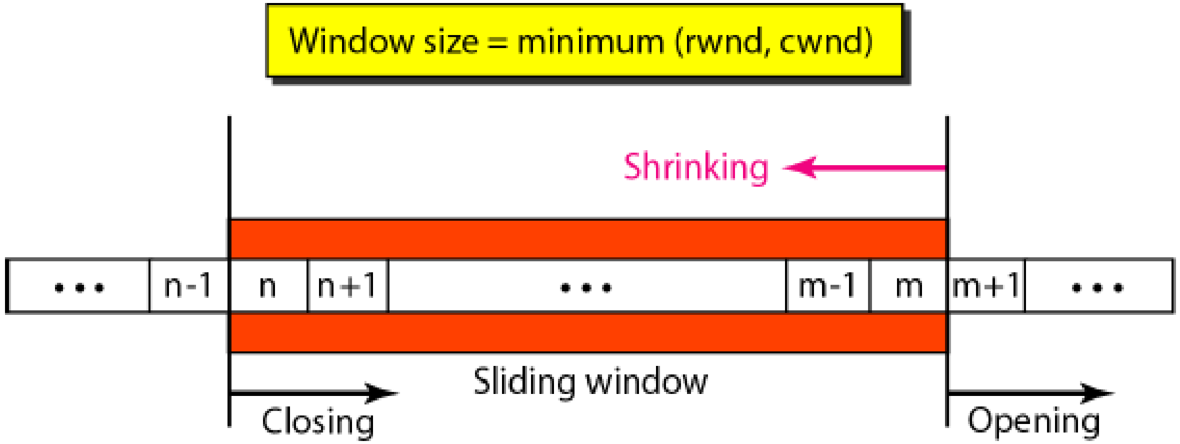


Selective-Repeat at Data Link Layer



TCP: Flow Control

- The **window spans a portion of the buffer** containing **bytes received** from the **process.**
- The **bytes inside the window** are the **bytes** that **can be in transit;** they **can be sent** without worrying about **acknowledgment**.
- The **imaginary window** has **two walls**: one **left** and one **right.**
- The **window** is ***opened, closed,* or *shrunk***. These **three activities**, are in the **control of the receiver** (and **depend on congestion** in the **network**), **not** the **sender.**



Flow Control

- **Opening a window**
  - Means **moving the right wall to the right**.
  - This allows **more new bytes** in the **buffer** that are **eligible** for **sending.**
- **Closing the window**
  - Means **moving the left wall to the right**.
  - This means that **some bytes** have been **acknowledged** and the **sender** need not worry about them anymore.
- **Shrinking the window**
  - Means **moving the right wall to the left.**
  - It means **revoking** the **eligibility** of **some bytes** for **sending.**

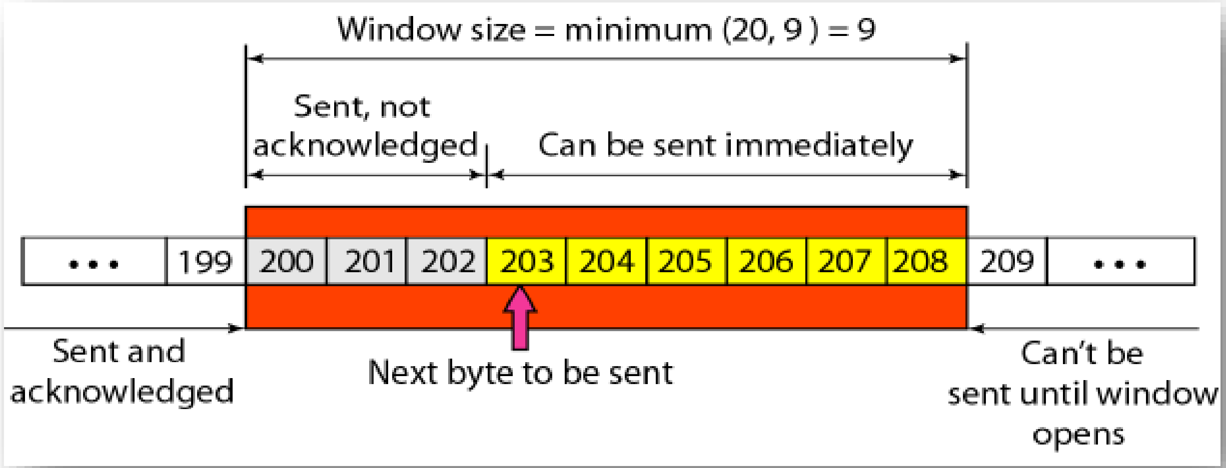
Flow Control

- The **size of the window** at **one end** is **determined** by the **lesser of two** values:
  - ***receiver window (rwnd)***
  - ***congestion window (cwnd).***
- The ***receiver window*** is the value **advertised** by the **opposite end** in a **segment** containing **acknowledgment.**
- It is the **number of bytes** the **other end** can **accept** before its **buffer overflows** and **data** are **discarded.**
- The **congestion window** is a value **determined** by the **network** to **avoid congestion**.

Flow Control

Example of a Sliding window.

- The **sender** has sent **bytes** up to **202**. Let **cwnd** is **20.**
- The **receiver** has sent an **acknowledgment number** of **200** with an ***rwnd*** of **9 bytes**.
- The **size** of the **sender window** is the **minimum of *rwnd* and *cwnd*,** or **9 bytes.**
- Bytes **200** to **202** are **sent,** but **not acknowledged.**
- Bytes **203** to **208** **can be sent** without worrying about **acknowledgment.**
- Bytes **209** and **above** **cannot be sent.**



Error Control in TCP

- **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**.
- **TCP** provides **reliability** using **error control**.
- **Error control** includes **mechanisms** for **detecting corrupted segments**, **lost segments**, **out-of-order segments**, and **duplicated segments.**
- **Error control** also includes a **mechanism for correcting errors** after they are **detected**.
- **Error detection** and **correction** in **TCP** is **achieved** through the use of **three simple tools:**
  - ***Checksum, Acknowledgment*** and ***Retransmission*****.**

Error Control in TCP

1. Checksum

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

2. 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**.
- **ACK segments** do not **consume sequence numbers** and are **not acknowledged**.

Error Control in TCP

3. Retransmission

- The **heart** of the **error control mechanism** is the **retransmission** of **segments**.
- When a **segment** is **corrupted**, **lost**, or **delayed**, it is **retransmitted.**
- In **modern implementations**, a **segment** is **retransmitted** on **two occasions**:
  - when a **Retransmission** **timer expires** or
  - when the **Sender receives** **three duplicate ACKs**.
- Note that **no retransmission** occurs for **segments** that do **not consume** **sequence numbers.**
- In particular, there is **no retransmission** for an **ACK segment**.

Retransmission After RTO

- A **recent implementation** of **TCP** maintains one **retransmission time-out (RTO) timer** for **all outstanding** (sent, but **not acknowledged**) **segments.**
- When the **timer matures**, the **earliest outstanding segment** is **retransmitted** .
- Note that **no time-out timer** is **set** for a **segment** that **carries** only an **acknowledgment,** which means that no such **segment** is **resent.**
- The value of **RTO is dynamic** in **TCP** and is **updated** based on the **round-trip time (RTT)** of **segments.**
- An **RTT** is the **time** needed for a **segment** to **reach** a **destination** and for an **acknowledgment** to be **received.**

Retransmission After Three Duplicate ACK Segments

- The previous **rule** about **retransmission** of a **segment** is **sufficient** if the **value of RTO** is **not very large.**
- Sometimes, however, **one segment is lost** and the **receiver receives so many out-of-order segments** that they **cannot be saved (limited buffer size).**
- To **alleviate** this **situation**, most implementations today follow the **three-duplicate-ACKs rule** and **retransmit** the **missing segment immediately.**
- This **feature** is referred to as **Fast Retransmission.**

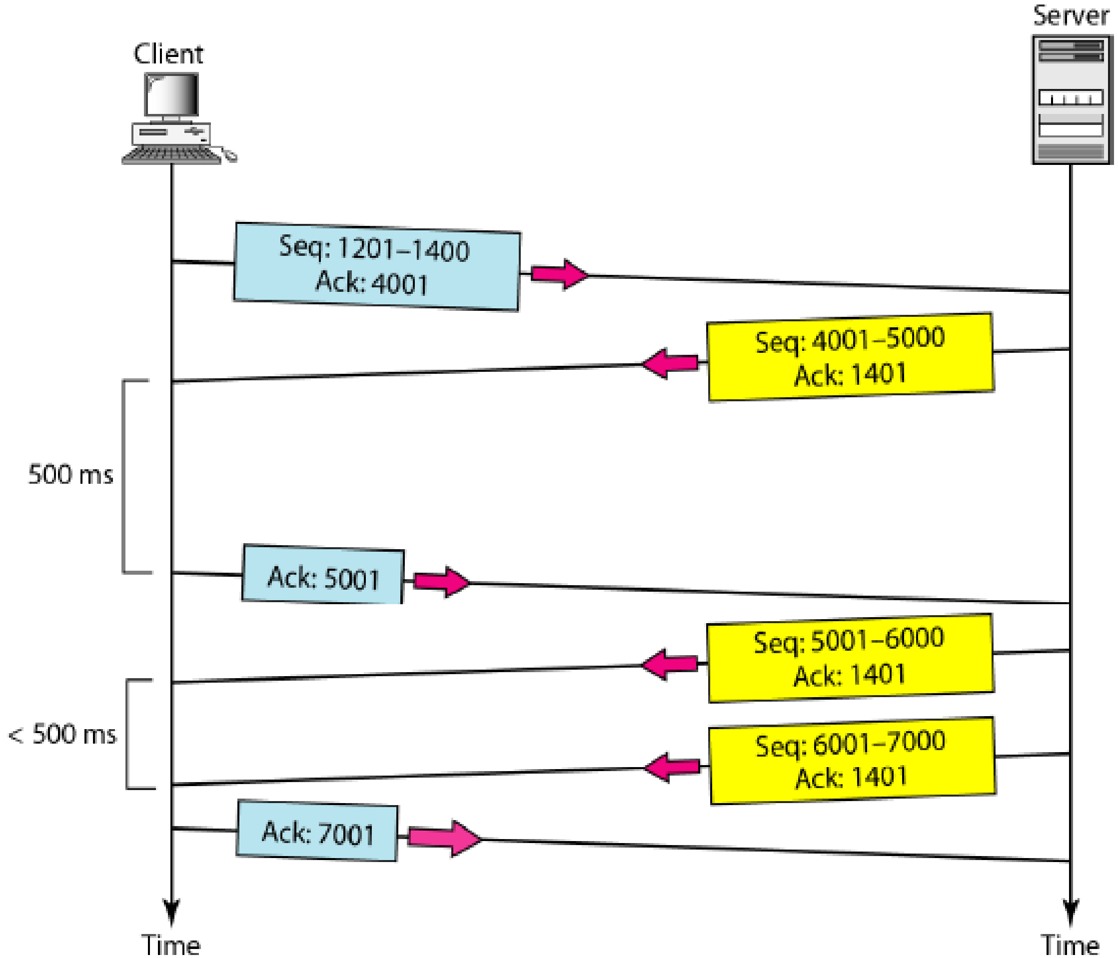
Out-of-Order Segments

- When a **segment** is **delayed, lost,** or **discarded**, the **segments** following that segment **arrive out of order**.
- Originally, **TCP** was designed to **discard all out-of-order segments**, resulting in the **retransmission** of the **missing segment** and the **following segments.**
- Most implementations today **do not discard** the **out-of-order segments**.
- They **store** them **temporarily** and **flag** them as **out-of-order segments** until the **missing segment arrives.**
- Note, however, that the **out-of-order segments** are **not delivered** to the **process.**
- **TCP guarantees** that **data** are **delivered** to the **process in order**.

Some Scenarios of TCP Operation: Normal Operation

- The **client TCP** sends **one segment**; the **server TCP** sends **three.**
- There are **data** to be **sent,** so the **segment** displays the **next byte** expected.
- When the **client receives** the **first segment** from the **server**, it does not have any more data to send; it sends only **an ACK segment**.
- However, the **acknowledgment** needs to be **delayed** for **500 ms** to see if any more segments arrive.
- When the **timer matures**, it **triggers an acknowledgment**.
- When the **next segment** arrives, another **acknowledgment** timer is set.
- However, before it matures, the **third segment** arrives.
- The **arrival** of the **third segment** triggers another acknowledgment**(cumulative ACK).**

TCP: Normal Operation



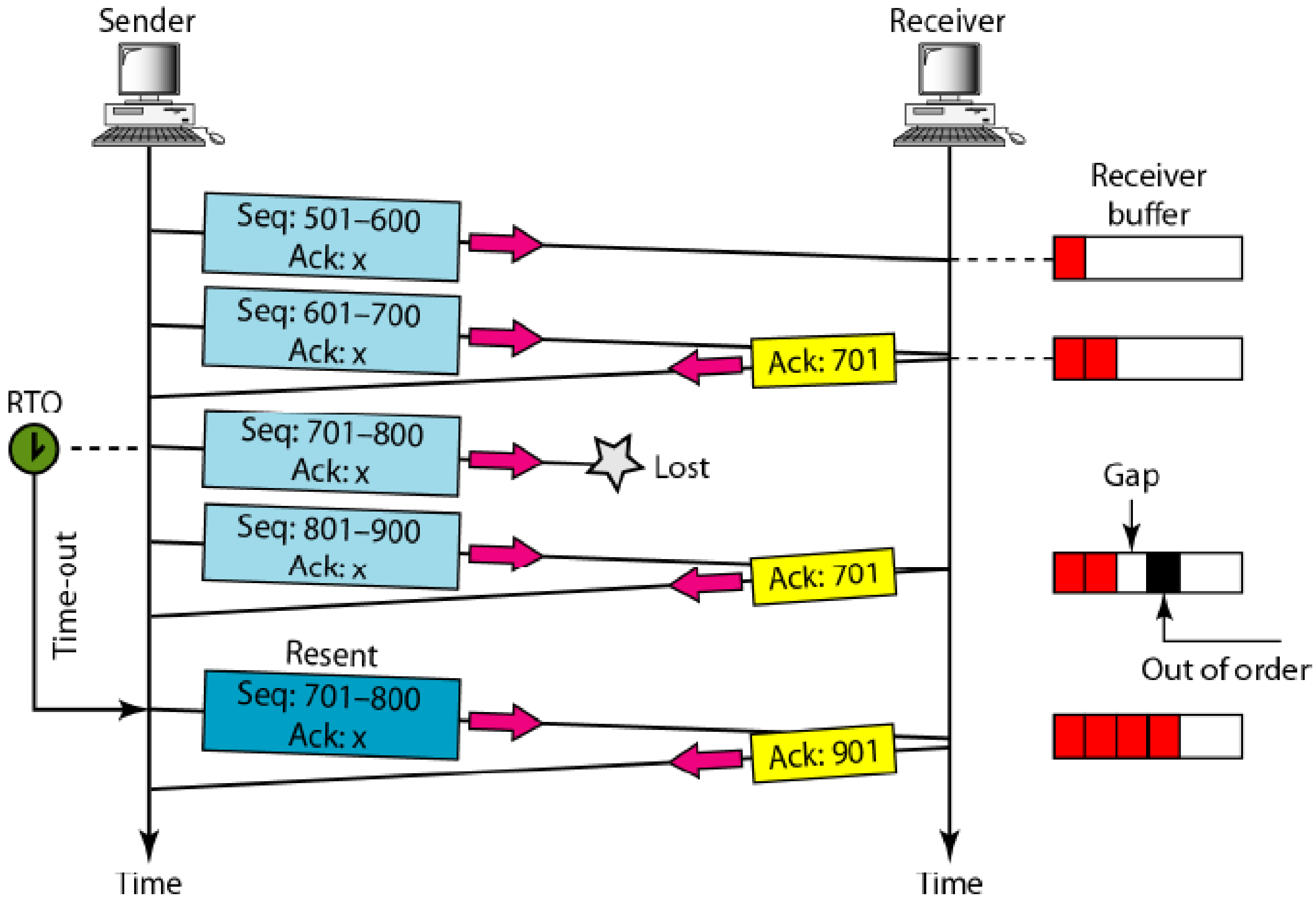
Lost Segment

- A **lost segment** and a **corrupted segment** are **treated** the **same way** by the **receiver.**
- A **lost segment** is **discarded** somewhere in the network; a **corrupted segment** is **discarded** by the **receiver itself**. Both are considered **lost**.
- We are assuming that **data transfer** is **unidirectional:** one site is sending, the other is receiving.
- In our scenario, the **sender** sends **segments 1** and **2**, which are **acknowledged immediately** by an **ACK.**
- **Segment 3**, however, is **lost.**
- The **receiver** receives **segment 4,** which is **out of order**.
- The **receiver stores** the **data** in the **segment** in its **buffer** but **leaves a gap** to indicate that there is **no continuity** in the **data.**

Lost Segment

- The **receiver** immediately **sends** an **acknowledgment** to the **sender**, displaying the **next byte** it **expects.**
- Note that the **receiver** **stores bytes 801 to 900**, but **never delivers** **these bytes** to the **application** until the **gap is filled**.
- We have shown the **timer** for the **earliest outstanding segment**.
- The **timer** for this definitely **runs out** because the **receiver never sends** an **acknowledgment** for **lost** or **out of-order segments.**
- When the **timer matures**, the **sending TCP** **resends segment 3,** which **arrives** this **time** and is **acknowledged** properly.
- Note that the **value** in the **second** and **third acknowledgments** differs according to the corresponding rule.

TCP: Lost Segment



Fast Retransmission

- This **scenario** is used when the **RTO has a higher value.**
- When the **receiver** receives the **fourth, fifth**, and **sixth segments**, it **triggers** an **acknowledgment.**
- The **sender** receives **four acknowledgments** with the **same value** (three duplicates).
- Although the **timer** for **segment 3** has **not matured** yet, the **fast transmission** requires that **segment 3**, the segment that is expected by all these acknowledgments, be **resent immediately.**
- Note that only **one segment** is **retransmitted** although **four segments** are **not acknowledged.**
- When the **sender** receives the **retransmitted ACK**, it knows that the **four segments** are **safe and sound** because **acknowledgment** is **cumulative.**

Fast Retransmission