

# Client-Server Model

# Client-Server Model

- **Client** application program running on the **local machine** requests a service from another application program – server – running on the **remote machine**.
- Commonly server provides service to any client, not a particular client
- Generally, a **client** application program that requests a service should run **only when** it is **needed**. A **server** program providing service should run **all the time**, as it does not know when its services will be needed.

# Client-Server Model

- A **client opens** the communication channel using IP address of the remote host and the port address of the specific server program running on the host – **Active open**.
- Request-response may be repeated several times, the process is **finite**.
- The client closes the communication channel with an **Active close**.

# Client-Server Model

- A server program opens its door for incoming requests from clients but never initiates a service unless explicitly requested – **Passive open**.
- A **server** program is **infinite** – runs unless a problem occurs.
- **Concurrency** in client: two or more clients can run at the same time on a machine – current trend, **alternatively**: one client must start, run, and terminate before another client may start (**iterative**).

# Client-Server Model

- Concurrency in server: An iterative server can process only one request at a time whereas a concurrent server can process many requests at the same time – share its time.
- Connectionless iterative server: the ones that use UDP are iterative, server uses one single port, arriving packets wait in line,
- Connection oriented concurrent server: the ones that use TCP are normally concurrent,

# Client-Server Model

- A **connection** is established between server and each client, remains **open until** the entire stream is processed
  - Each connection requires a port and **many** connections may remain open simultaneously,
  - Server can use only **one** well-known port.
  - **Solution:** use additional *ephemeral* ports
  - Client makes **request through** the well-known port, once the connection is made, server assigns a temporary port to this connection.
-

# Client-Server Model

- => well-known port is **free** to receive requests for additional connections
- A **program** is code (in UNIX) and defines all the variables and actions to be performed on those variables.
- A **process** is an instance of a program.
- An OS creates a process when executing a program, several **processes** can be created from one program running **concurrently**.

# Client-Server

1. \_ can provide a service.

- a) An iterative server
- b) A concurrent server
- c) A client
- d) (a) and (b)

2. The client program is \_ as it terminates after it has been served.

- a) Active
- b) Passive
- c) Finite
- d) Infinite



# Client-Server

3. A connection oriented concurrent server uses \_ ports.

- a) Ephemeral
- b) Well-known
- c) Active
- d) (a) and (b)

4. Machine A requests service X from machine B. Machine B requests service Y from machine A. What is the total number of application programs required?

- a) One
- b) Two
- c) Three
- d) Four

# Client-Server

5. A server program, once it issues \_\_, waits for clients to request its service.

- a) An active open
- b) A passive open
- c) An active request
- d) A finite open

6. \_\_ processes requests one at a time.

- a) An iterative client
- b) An iterative server
- c) A concurrent client
- d) A concurrent server

# Client-Server

7. In a connection oriented concurrent server, the \_ is used for connection only.

- a) Ephemeral port
- b) Well-known port
- c) Infinite port
- d) (a) and (b)

8. \_ is an instance of a \_.

- a) Process; service
- b) Program; process
- c) Process; program
- d) Structure; process

# Socket

- **Socket**: is a construct that supports network input/output.
- An application **creates a socket** when it needs a connection to a network.
- It then establishes a connection to a **remote application** via the socket.
- Communication is achieved by **reading data** from the socket and **writing data** to it.
- Socket acts as an endpoint.

# Socket

- A socket is defined in the OS as a **structure**.
- Simplified socket structure with five fields:
  - **Family**: defines the protocol group (IPv4, IPv6, UNIX domain protocols)
  - **Type**: defines the exchange-type (stream, packet, raw)
  - **Protocol**: set to zero for TCP/ UDP
  - **Local address**: combination of local IP and application port address.
  - **Remote address**: combination of remote IP and application port address.

# Socket

- Stream socket: to be used with connection oriented protocol (TCP)
- Datagram socket: to be used with connectionless protocol (UDP)
- Raw socket: some protocols (ICMP) directly use the service of IP. Raw sockets are used in these applications.

# Socket

- **Connectionless iterative server:** receives a request packet from UDP, processes the request, gives response to the UDP to send to the client,
- Packets are stored in a queue, and processed in order of arrival.
- **Server function:**
  - **Create** a socket (asks the OS)
  - **Bind** (asks the OS to enter information in the socket related to the server – server socket binding)

# Socket

- Repeats steps **infinitely**:
  - a) **receive** a request (asks the OS to wait for a request to this socket and receive it)
  - b) **process** (by itself)
  - c) **send** (response sent to the client)
- Client function:
  - **Create** a socket (asks the OS)
  - Repeats steps as long as it has requests:



# Socket

- a) **send** a request (asks the OS)
- b) **receive** (asks the OS to wait for a response and deliver it when it has arrived)
  - **Destroy** (asks the OS to destroy the socket once requests are **exhausted**)

# Socket

- **Connection oriented concurrent server:**
- A connection for each client, one buffer for each connection
- *Parent and child server:* server running infinitely accepting connections is parent, after establishing connection, parent creates **child-server** and ephemeral ports to handle the client.

# Socket

- **Server function:**
  - **Create** a socket (asks the OS)
  - **Bind** (asks the OS to enter information in the socket related to the server – server socket binding)
  - **Listen** (asks the OS to be passive and listen to the client)
  - Repeats steps infinitely:
    - a) **Create** a child (temporary child process is assigned to serve the client, parent is free)
    - b) **create** a new socket (to be used by the child process)

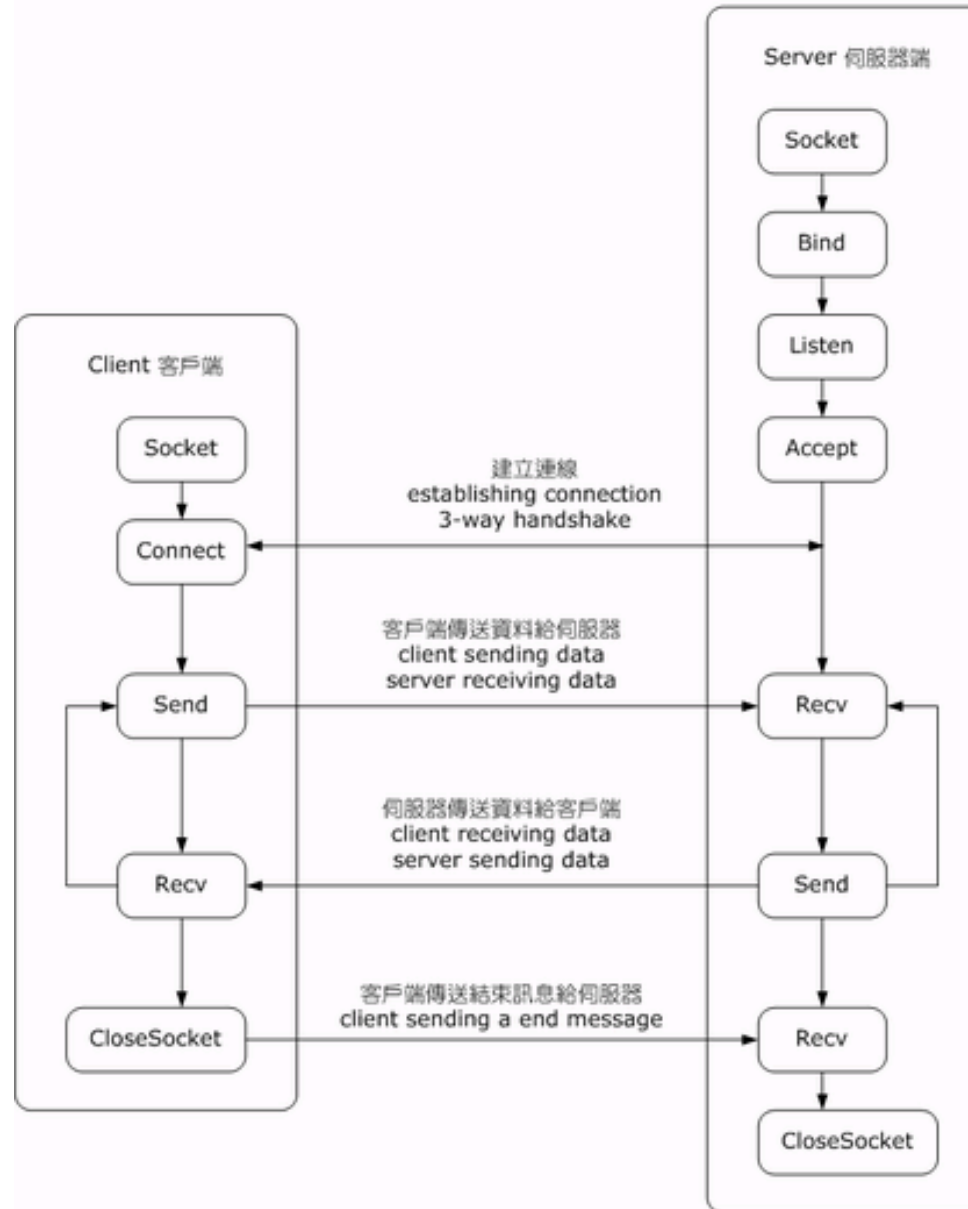
# Socket

- c) **Repeat** (the following steps as long as the child has requests from the client)
  - **Read** (child reads a stream of bytes from the connection)
  - **Process** (the child processes the stream of bytes)
  - **write** (the child writes the result as a stream of bytes to the connection)
- d) **Destroy** (asks the OS to destroy the temporary socket once the client has been served)

# Socket

- Client function:
  - Create a socket
  - Connect (asks the OS to make a connection)
  - Repeats steps as long as it has data to send:
    - a) Write (sends a stream of bytes to the server)
    - b) Read (receives a stream of bytes from the server)
  - Destroy (asks the OS to destroy the socket once it has finished)

TCP Socket 基本流程圖  
TCP Socket flow diagram



# Socket

1. The `_socket` is used with a connection oriented protocol.

- a) Stream
- b) Datagram
- c) Raw
- d) Remote

2. The `_socket` is used with a connectionless protocol.

- a) Stream
- b) Datagram
- c) Raw
- d) Remote

# Socket

3. The `_socket` is used with a protocol that directly uses the services of IP.
  - a) Stream
  - b) Datagram
  - c) Raw
  - d) Remote
4. A `_server` serves multiple clients, handling one request at a time.
  - a) Connection-oriented iterative
  - b) Connection-oriented concurrent
  - c) Connectionless iterative
  - d) Connectionless concurrent



# Socket

5. A \_ server serves multiple clients simultaneously.

- a) Connection-oriented iterative
- b) Connection-oriented concurrent
- c) Connectionless iterative
- d) Connectionless concurrent

# Socket

- Data structures:

`sockaddr_in`

`sin_family`: 16-bit integer specifying protocol

`sin_port`: 16-bit field specifying port number  
(application)

`s_addr`: 32-bit Internet address

`hostent`

`h_name`: character string of text address of host

# Socket

h\_alias: alternative names

h\_addrtype: type of address

h\_length: Address length

h\_addr\_list: additional addresses

# Socket

## Socket Commands:

‘**socket**’ – creates a socket

‘**gethostbyname**’ – returns a host IP address  
corresponding to a name

‘**gethostname**’ – returns the host name

‘**connect**’ – requests a connection with a remote  
socket

‘**bind**’ – assigns an address and port number to a  
socket

# Socket

## Socket Commands:

‘**listen**’ – server is ready for connection requests and listens for them

‘**accept**’ – accepts a connection request over a socket

‘**send**’ – sends data through a socket

‘**recv**’ – receives data from a socket

‘**close**’ – closes a socket

# Socket

# Does the client or the server or both usually execute each of the following socket commands?

- |           |            |         |           |
|-----------|------------|---------|-----------|
| a. Socket | b. connect | c. bind | d. accept |
| e. Listen | f. send    | g. recv | h. close  |

# Socket

**Client server example:** outline / framework of  
socket related calls

‘client and server run concurrently on different  
machines’

# Socket

## Client:

socket (creates a socket)

.....

gethostbyname (maps remote host name to an IP  
address)

.....

Connect (issues a connection request to a  
specific server on the remote host)

.....



# Socket

exchange information using 'send' and 'recv'  
commands

.....

close (terminate connection)

**Server:**

socket (creates a socket)

.....

gethostname (get the local host name)

# Socket

gethostbyname (maps the host name to an IP address)

.....

bind ( specifies the IP address and the port number and associates them with the socket)

.....

listen (puts socket in passive mode; ready to accept request)

.....

# Socket

accept (accepts a connection request)

.....

exchange information using 'send' and 'recv'  
commands

.....

close (terminates current connection)