SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be **University** *u/s 3 of UGC Act, 1956*

18CSC302J- Computer Networks

Unit II

TCP Client Server Program
Input, Output Processing Module
UDP Client Server Program
UDP Control block table & Module
UDP Input & Output Module

Syllabus - Unit II

- Byte ordering
- Byte ordering conversion functions
- System calls
- Sockets
- System calls used with Sockets
- Iterative and concurrent server
- Socket Interface
- Structure and Functions of Socket
- Remote Procedure Call
- RPC Model, Features
- **TCP Client Server Program**
- **Input, Output Processing Module**
- **UDP Client Server Program**
- **UDP Control block table & Module**
- **UDP Input & Output Module**
- SCTP Sockets
- SCTP Services and Features, Packet Format
- SCTP Client/Server

Headers Files

- To use socket interface functions, we need a set of header files.
- We define this header file in a separate file, which we name "headerFiles.h".
- We include this file in our programs to avoid including long lists of header files.
- Not all of these header files may be needed in all programs, but it is recommended to include all of them in case.

```
// "headerFiles.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/wait.h>
```

Client-Sever Paradigm

- The purpose of a network, or an internetwork, is to provide services to users: A user at a local site wants to receive a service from a computer at a remote site.
- One way to achieve this purpose is to run two programs.
- A local computer runs a program to request a service from a remote computer; the remote computer runs a program to give service to the requesting program.
- This means that two computers, connected by an internet, must each run a program, one to provide a service and one to request a service.
- To enable communication between two application programs, one running at the local site, the other running at the remote site.

Client-Sever Paradigm-Questions

- **Should both application programs be able to request services and provide services or should the application programs just do one or the other?**
  - One solution is to have an application program, called the client, running on the local machine, request a service from another application program, called the server, running on the remote machine.
  - In other words, the tasks of requesting a service and providing a service are separate from each other.
  - An application program is either a requester (a client), or a provider (a server).
  - In other words, application programs come in pairs, client and server, both having the same name.

Client-Sever Paradigm-Questions

- **Should a server provide services only to one specific client or should the server be able to provide services to any client that requests the type of service it provides?**
  - The most common solution is a server providing a service for any client that needs that type of service, not a particular one.
  - In other words, the server-client relationship is one-to-many.
- **Should a computer run only one program (client or server)?**
  - The solution is that any computer connected to the Internet should be able to run any client program if the appropriate software is available.
  - The server programs need to be run on a computer that can be continuously running as we will see later.

Client-Sever Paradigm-Questions

- **When should an application program be running? All of the time or just when there is a need for the service?**
  - Generally, a client program, which requests a service, should run only when it is needed.
  - The server program, which provides a service, should run all the time because it does not know when its service will be needed.
- **Should there be only one universal application program that can provide any type of service a user wants? Or should there be one application program for each type of service?**
  - In TCP/IP, services needed frequently and by many users have specific client-server application programs.

Client-Sever Paradigm-Questions

- For example, we have separate client-server application programs that allow users to access files, send e-mail, and so on.

- For services that are more customized, we should have one generic application program that allows users to access the services available on a remote computer.
  - For example, we should have a client-server application program that allows the user to log onto a remote computer and then use the services provided by that computer.

Server and Client

Server

- A server is a program running on the remote machine providing service to the clients.
- When it starts, it opens the door for incoming requests from clients, but it never initiates a service until it is requested to do so.
- A server program is an infinite program. When it starts, it runs infinitely unless a problem arises.
- It waits for incoming requests from clients.
- When a request arrives, it responds to the request, either iteratively or concurrently as we will see shortly.

Server and Client

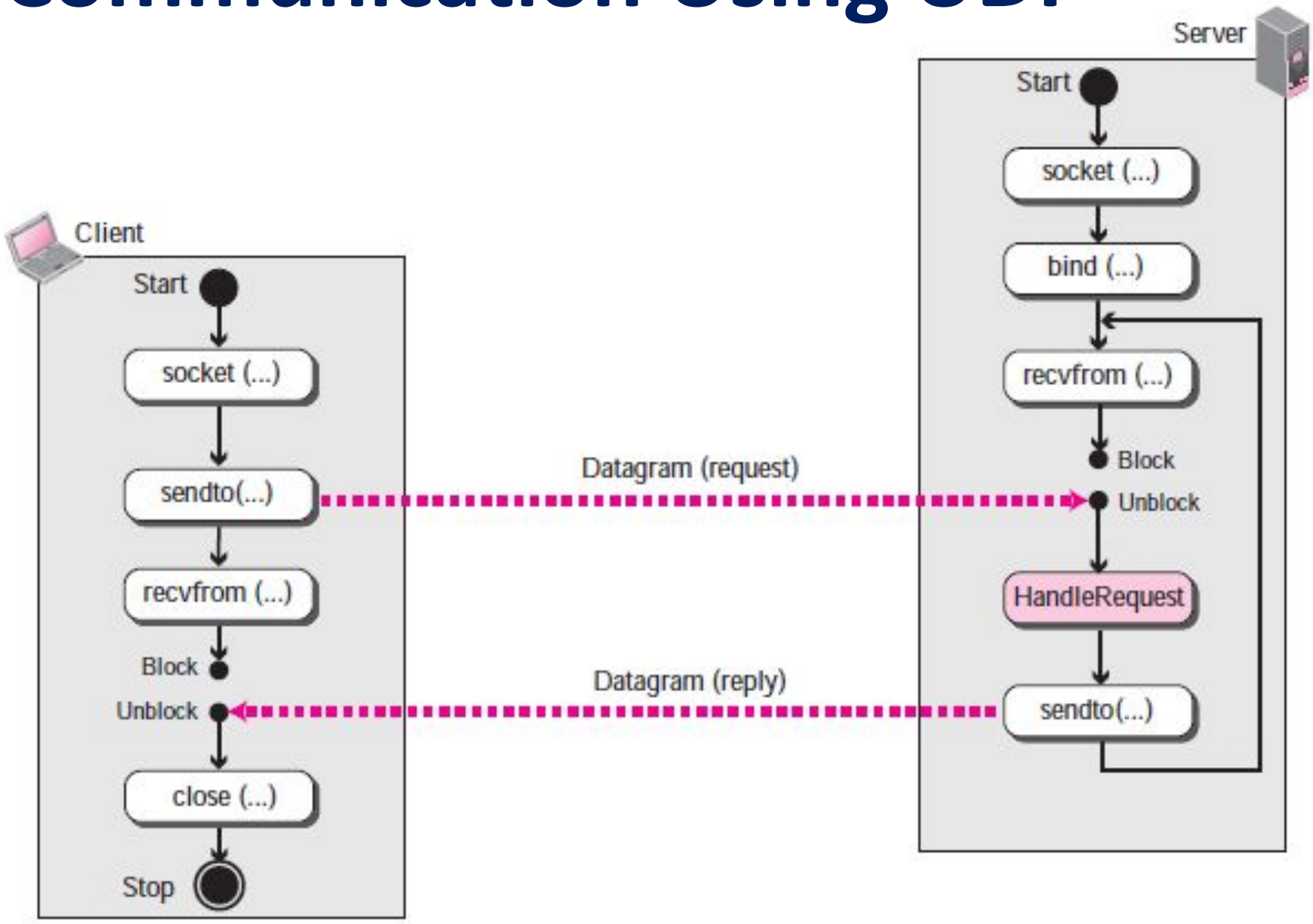
Client

- A client is a program running on the local machine requesting service from a server.
- A client program is finite, which means it is started by the user (or another application program) and terminates when the service is complete.
- Normally, a client opens the communication channel using the IP address of the remote host and the well-known port address of the specific server program running on that machine.
- After a channel of communication is opened, the client sends its request and receives a response.
- Although the request-response part may be repeated several times, the whole process is finite and eventually comes to an end.

UDP

1. Communication Using UDP

Connectionless iterative communication using UDP



Echo Server Program using the Service of UDP

```
01 // UDP echo server program
02 #include "headerFiles.h"
03
04 int main (void)
05 {
06       // Declaration and definition
07       int sd;                                  // Socket descriptor
08       int nr;                                  // Number of bytes received
09       char buffer [256];                       // Data buffer
10       struct sockaddr_in serverAddr;           // Server address
11       struct sockaddr_in clientAddr;           // Client address
12       int clAddrLen;                           // Length of client Address
13       // Create socket
14       sd = socket (PF_INET, SOCK_DGRAM, 0);
15       // Bind socket to local address and port
16       memset (&serverAddr, 0, sizeof (serverAddr));
17       serverAddr.sin_family = AF_INET;
18       serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);        // Default address
19       serverAddr.sin_port = htons (7)     // We assume port 7
20       bind (sd, (struct sockaddr*) &serverAddr, sizeof (serverAddr));
21       // Receiving and echoing datagrams
22       for ( ; ; )         // Run forever
23       {
24            nr = recvfrom (sd, buffer, 256, 0, (struct sockaddr*)&clientAddr, &clAddrLen);
25            sendto (sd, buffer, nr, 0, (struct sockaddr*)&clientAddr, sizeof(clientAddr));
26       }
27 } // End of echo server program
```

UDP-Server Process

- The server process starts first.
- The server process calls the socket function to create a socket.
- It then calls the bind function to bind the socket to its well-known port and the IP address of the computer on which the server process is running.
- The server then calls the recvfrom function, which blocks until a datagram arrives.
- When the datagram arrives, the recvfrom function unblocks, extracts the client socket address and address length from the received datagram, and returns them to the process.
- The process saves these two pieces of information and calls a procedure (function) to handle the request.
- When the result is ready, the server process calls the sendto function and uses the saved information to send the result to the client that requested it.
- The server uses an infinite loop to respond to the requests coming from the same client or different clients.

Echo Client Program using the Service of UDP

```
01  // UDP echo client program
02  #include "headerFiles.h"
04
04  int main (void)
05  {
06      // Declaration and definition
07      int sd;                          // Socket descriptor
08      int ns;                          // Number of bytes send
09      int nr;                          // Number of bytes received
10      char buffer [256];               // Data buffer
11      struct sockaddr_in serverAddr;   // Socket address
11      // Create socket
12      sd = socket (PF_INET, SOCK_DGRAM, 0);
13      // Create server socket address
14      memset (&servAddr, 0, sizeof(serverAddr));
15      servAddr.sin_family = AF_INET;
16      inet_pton (AF_INET, "server address", &serverAddr.sin_addr);
17      serverAddr.sin_port = htons (7);
18      // Send and receive datagrams
19      fgets (buffer, 256, stdin);
20      ns = sendto (sd, buffer, strlen (buffer), 0,
21          (struct sockaddr)&serverAddr, sizeof(serveraddr));
22      recvfrom (sd, buffer, strlen (buffer), 0, NULL, NULL);
23      buffer [nr] = 0;
24      printf ("Received from server:");
25      fputs (buffer, stdout);
26      // Close and exit
27      close (sd);
28      exit (0);
29  } // End of echo client program
```

UDP-Client Process

- The client process is simpler.
- The client calls the socket function to create a socket.
- It then calls the sendto function and pass the socket address of the server and the location of the buffer from which UDP can get the data to make the datagram.
- The client then calls a recvfrom function call that blocks until the reply arrives from the server.
- When the reply arrives, UDP delivers the data to the client process, which make the recv function to unblock and deliver the data received to the client process.
- Note that we assume that the client message is so small that it fits into one single datagram.
- If this is not the case the two function calls, sendto and recvfrom, need to be repeated
- However, the server is not aware of multidatagram communication; it handles each request separately.

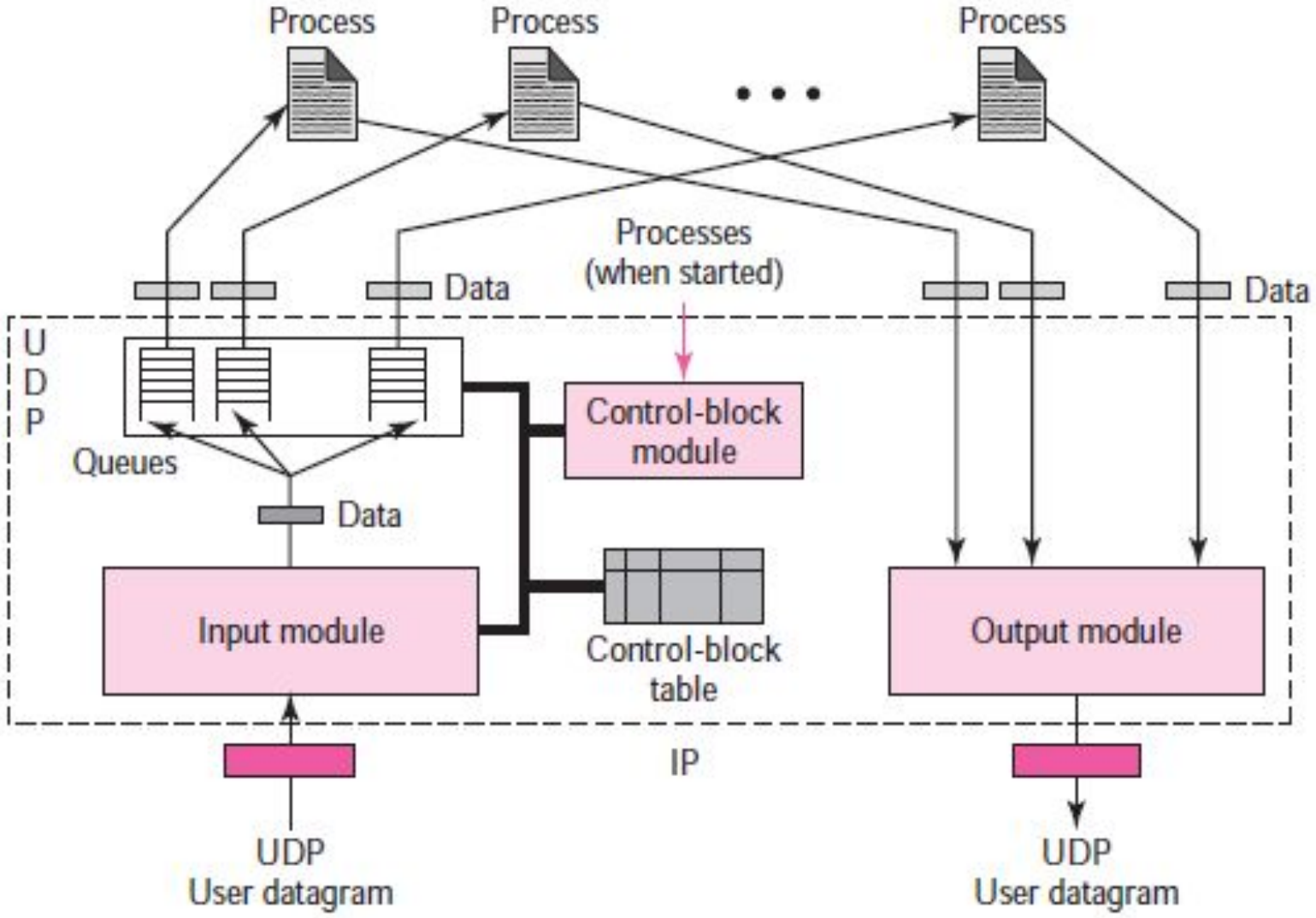
2. UDP Package

- The UDP package involves five components: a control-block table, input queues, a control-block module, an input module, and an output module.

- **Control-Block Table**
  - UDP has a control-block table to keep track of the open ports.
  - Each entry in this table has a minimum of four fields: the state, which can be FREE or IN-USE, the process ID, the port number, and the corresponding queue number.

- **Input Queues**
  - Our UDP package uses a set of input queues, one for each process. In this design, we do not use output queues.

UDP Design


Process
Process
Process
Processes
(when started)
Data
Data
U
D
P
Queues
Control-block
module
Data
Input module
Control-block
table
Output module
IP
UDP
User datagram
UDP
User datagram

UDP Package

Control-Block Module

- The control-block module is responsible for the management of the control-block table.
- When a process starts, it asks for a port number from the operating system.
- The operating system assigns well-known port numbers to servers and ephemeral port numbers to clients.
- The process passes the process ID and the port number to the control-block module to create an entry in the table for the process.

```
UDP_Control_Block_Module (process ID,
port number)
{
Search the table for a FREE entry.
if (not found)
Delete one entry using a predefined
strategy.
Create a new entry with the state IN-USE
Enter the process ID and the port number.
 Return.
} // End module
```

UDP Package

Input Module

- The input module receives a user datagram from the IP. It searches the control-block table to find an entry having the same port number as this user datagram.
- If the entry is found, the module uses the information in the entry to enqueue the data. If the entry is not found, it generates an ICMP message..

```
UDP_INPUT_Module (user_datagram)
{
Look for the entry in the control_block table
if (found)
{
Check to see if a queue is allocated
If (queue is not allocated)
 allocate a queue
else
enqueue the data
} //end if
else
{
Ask ICMP to send an "unreachable port" message
Discard the user datagram
} //end else
Return.
} // end module
```

UDP Package

Output Module

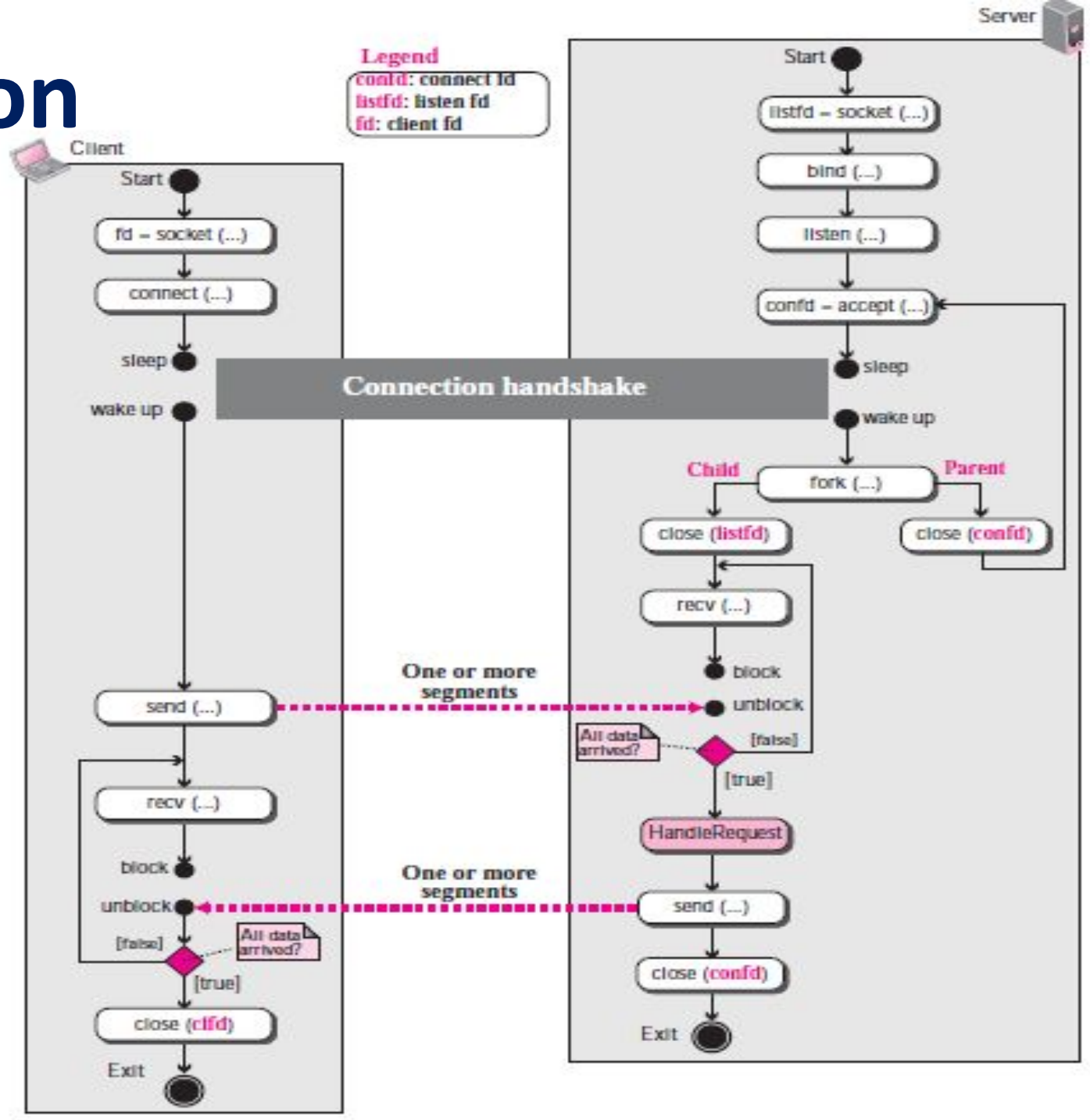
- The output module is responsible for creating and sending user datagrams.

```
UDP_OUTPUT_MODULE (Data)
{
    Create a user datagram
    Send the user datagram
    Return.
}
```

TCP

3. Communication Using TCP

Flow diagram for connection-oriented, concurrent communication



TCP-Server Process

- The server process starts first.
- It calls the socket function to create a socket, which we call the listen socket.
- This socket is only used during connection establishment.
- The server process then calls the bind function to bind this connection to the socket address of the server computer.
- The server program then calls the accept function.
- This function is a blocking function; when it is called, it is blocked until the TCP receives a connection request (SYN segment) from a client.
- The accept function then is unblocked and creates a new socket called the connect socket that includes the socket address of the client that sent the SYN segment.
- After the accept function is unblocked, the server knows that a client needs its service.
- To provide concurrency, the server process (parent process) calls the fork function.

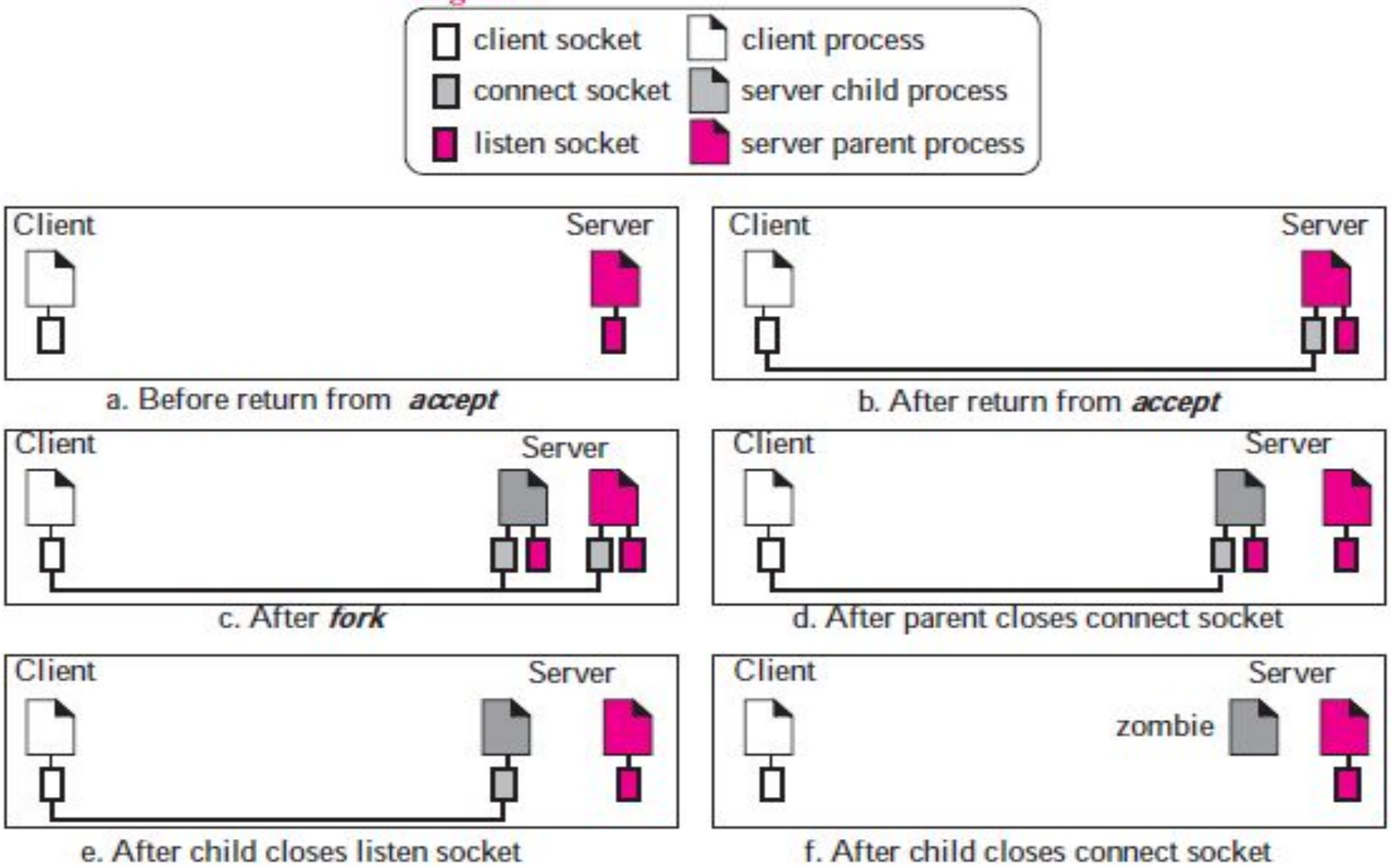
TCP-Server Process

- This function creates a new process (child process), which is exactly the same as the parent process.
- After calling the fork function, the two processes are running concurrently, but each can do different things.
- Each process now has two sockets: listen and connect sockets.
- The parent process entrusts the duty of serving the client to the hand of the child process and calls the accept function again to wait for another client to request connection.
- The child process is now ready to serve the client.
- It first closes the listen socket and calls the recv function to receive data from the client.
- The recv function, like the recvfrom function, is a blocking function; it is blocked until a segment arrives.
- The child process uses a loop and calls the recv function repeatedly until it receives all segments sent by the client.

TCP-Server Process

- The child process then gives the whole data to a function (handleRequest), to handle the request and return the result.
- The result is then sent to the client in one single call to the send function. We need to emphasize several points here.
- First, the flow diagram we are using is the simplest possible one.
- The server may use many other functions to receive and send data, choosing the one which is appropriate for a particular application.
- Second, we assume that size of data to be sent to the client is so small that can be sent in one single call to the send function; otherwise, we need a loop to repeatedly call the send function.
- Third, although the server may send data using one single call to the send function, TCP may use several segments to send the data.
- The client process, may not receive data in one single segment

Status of parent and child processes with respect to the sockets



TCP-Server Process

- The status of the parent and child process with respect to the sockets.
- Part a in the figure shows the status before the accept function returns.
- The parent process uses the listen socket to wait for request from the clients.
- When the accept function is blocked and returned (part b), the parent process has two sockets: the listen and the connect sockets.
- The client is connected to the connect socket.
- After calling the fork function (part c), we have two processes, each with two sockets.
- The client is connected to both processes.
- The parent needs to close its connect socket to free itself from the client and be free to listen to requests from other clients (part d).
- Before the child can start serving the connected client, it needs to close its listen
- socket so that a future request does not affect it (part e).
- Finally, when the child finishes serving the connected client, it needs to close its connect socket to disassociate itself from the client that has been served (part f).

TCP-Client Process

- The client process is simpler.
- The client calls the socket function to create a socket.
- It then calls the connect function to request a connection to the server.
- The connect function is a blocking function; it is blocked until the connection is established between two TCPs.
- When the connect function returns, the client calls the send function to send data
- to the server.
- We use only one call to the send function, assuming that data can be sent with one call.
- Based on the type of the application, we may need to call this function repeatedly (in a loop).

TCP-Client Process

- The client then calls the recv function, which is blocked until a segment arrives and data are delivered to the process by TCP.
- Note that, although the data are sent by the server in one single call to the send function, the TCP at the server site may have used several segments to send data.
- This means we may need to call the recv function repeatedly to receive all data.
- The loop can be controlled by the return value of the recv function.

TCP-client server example process

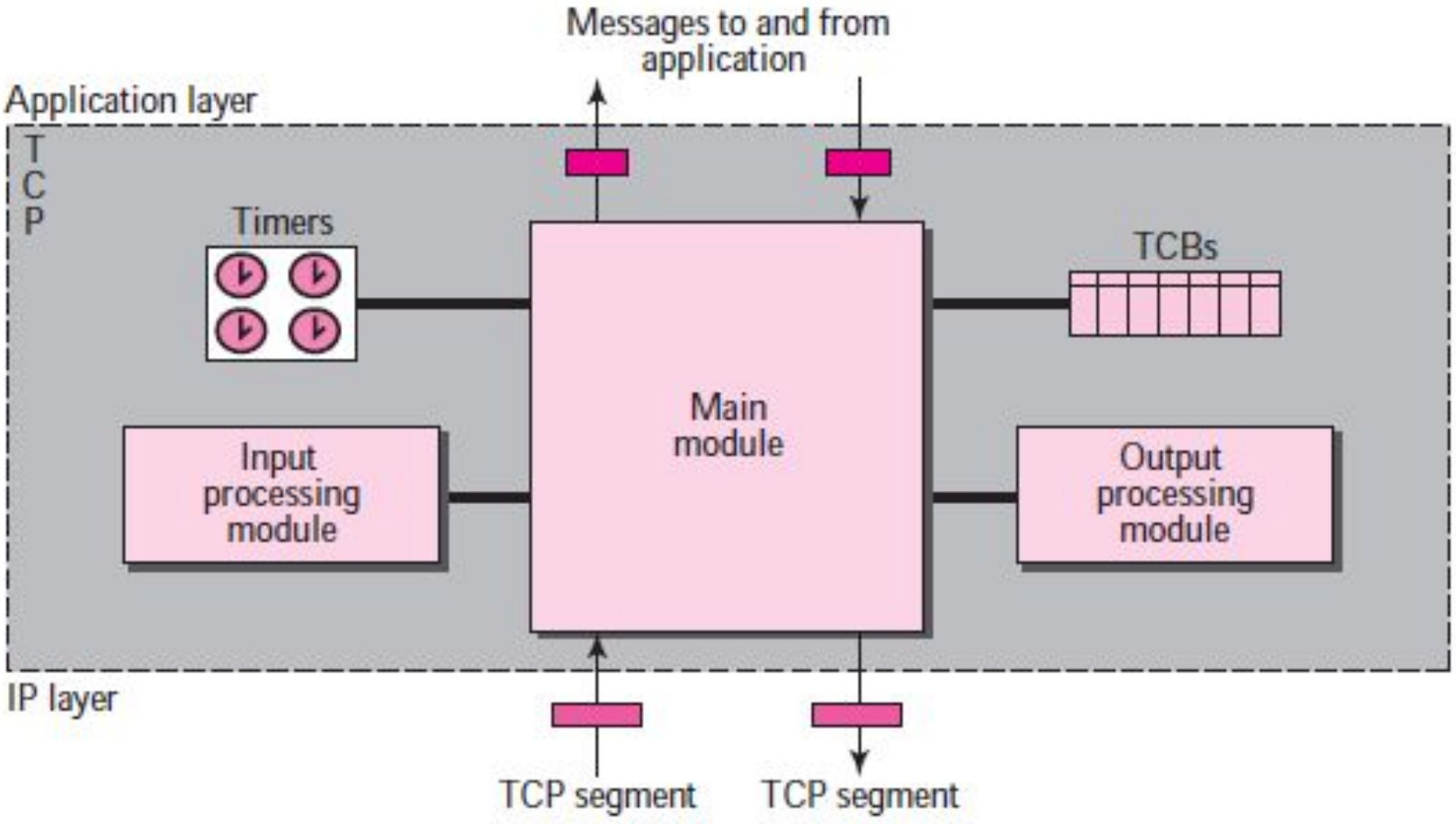
- Example:
  - TCP Client-Server program

Note: TCP example is given in hyperlink (as separate PDF document)

4. TCP Package

- TCP is a complex protocol
- TCP is a stream service, connection oriented protocol with an involved state transition diagram.
- It uses flow and error control.
- It is so complex because actual code includes tens of thousands of lines.
- TCP package involves tables called transmission control blocks, a set of timers, and three software modules: Main module, An input processing module, An output processing module

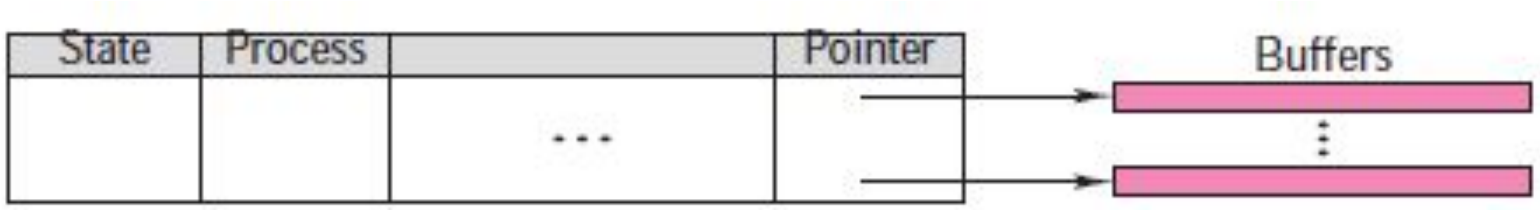
TCP Design


Messages to and from application
Application layer
TCP
Timers
TCBs
Main module
Input processing module
Output processing module
IP layer
TCP segment
TCP segment

TCP Package

Transmission Control Blocks (TCBs)

- TCP is a connection-oriented transport protocol.
- A connection may be open for a long period of time.
- To control the connection, TCP uses a structure to hold information about each connection. This is called a transmission control block (TCB).
- Because at any time there can be several connections, TCP keeps an array of TCBs in the form of a table. The table is usually referred to as the TCB



TCP Package

- **Timers**
  - Several timers TCP needs to keep track of its operations.
- **Input Processing Module**
  - The Input processing module handles all the details which is required for the process of data or an Acknowledgement received when TCP is in the ESTABLISHED STATE
  - This module sends an ACK if needed.
  - Takes care of the window size
  - Performs Error checking and so on.

TCP Package

- **Main module**

- The main module is invoked by an arriving TCP segment, a time-out event, or a message from an application program.

- This is a very complicated module because the action to be taken depends on the current state of the TCP.

- **Output Processing Module**

  - The output processing module handles all the details needed to send out data received from application program when TCP is in the ESTABLISHED STATE

  - This module handles retransmission time outs, persistent time outs and so on.

Thank You