CS321: Computer Networks

Introduction to Application Layer

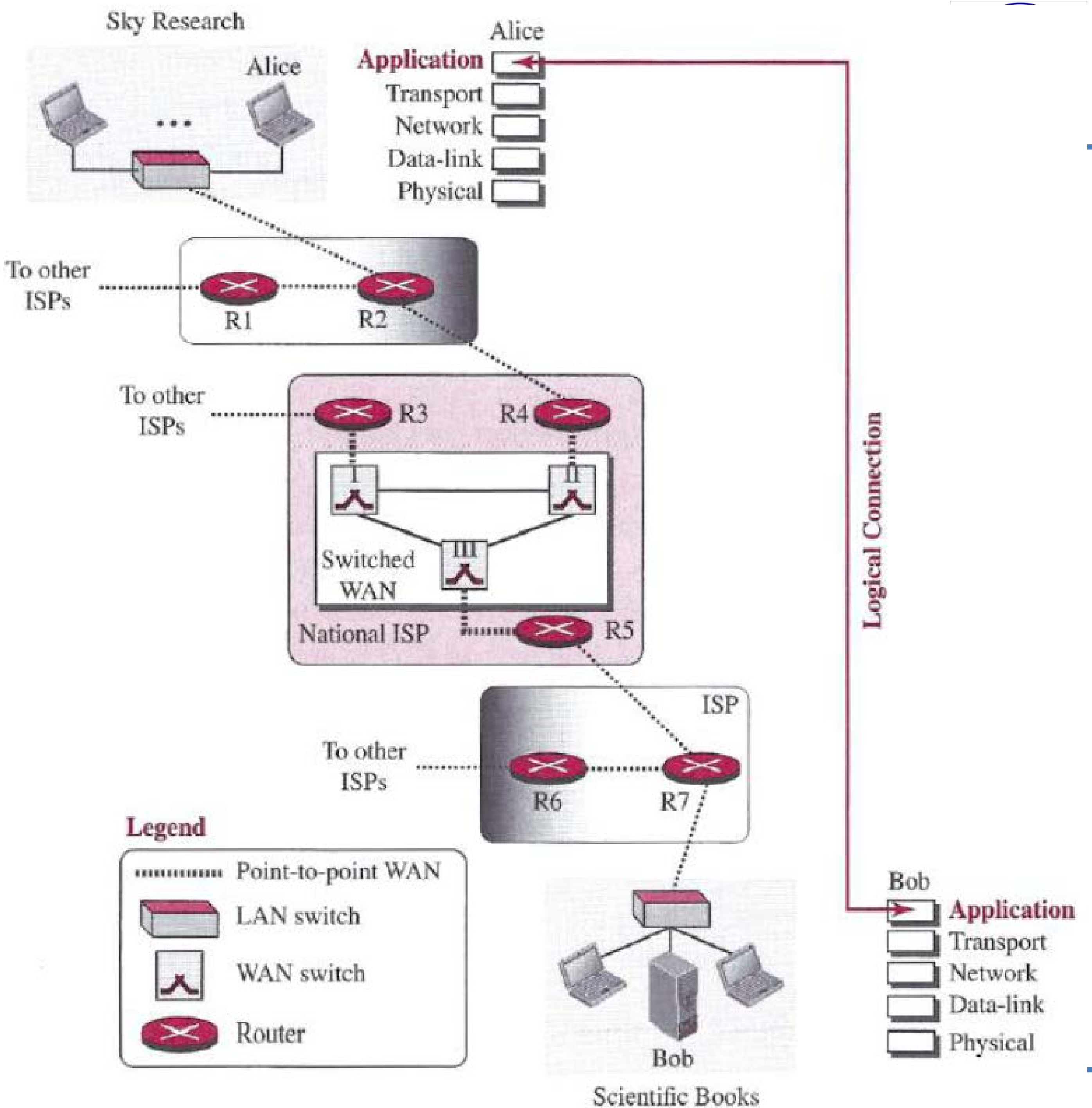
Dr. Manas Khatua
Assistant Professor
Dept. of CSE
IIT Jodhpur
E-mail: manaskhatua@iitj.ac.in

Basic

Application layer provides services to the user, and takes services from Transport layer.

Communication is provided using a logical connection.

The actual communication takes place through several physical devices (Alice, R2, R4, R5, R7, and Bob) and channels.



Providing Services

- Application layer provides services to the user, and takes services from Transport layer.
- The application layer, however, is somewhat different from other layers.
- The protocols can be removed from this layer easily as they only receives services from Transport layer but does not provide any service to that layer.
- The protocols used in the first four layers of the *TCP/IP* suite need to be standardized and documented. So, normally comes with Operating Systems (OS).
- There are several application-layer protocols that have been standardized and documented by the Internet authority.
  E.g., DHCP, SMTP, FTP, HTTP, TELNET

Application-Layer Paradigms

- We need two application programs to interact with each other.

- What the relationship should be between these two programs?
  - *client-server paradigm*
  - *peer-to-peer paradigm*

Client-Server Paradigm

- the service provider is an application program, called the server process; it runs continuously, waiting for another application program, called the client process, to make a connection through the Internet and ask for service.

- The server process must be running all the time; the client process is started when the client needs to receive service.

- Several traditional services are still using this paradigm, e.g., WWW, HTTP, FTP, SSH, E-mail, and so on.

- Problems:
  - the server should be a powerful computer
  - there should be a service provider willing to accept the cost and create a powerful server for a specific service

Peer-to-Peer Paradigm

- There is no need for a server process to be running all the time and waiting for the client processes to connect.
- The responsibility is shared between peers.
- E.g.: Internet telephony, BitTorrent, Skype, IPTV
- Advantages:
  - easily scalable and cost-effective in eliminating the need for expensive servers to be running and maintained all the time
- Challenges:
  - more difficult to create secure communication between distributed services

Client-Server Programming

- Runs two processes: a client and a server

- A client is a running program that initializes the communication by sending a request;
- A server is another application program that waits for a request from a client.

- A client program is started and stopped by the user whenever it requires.
- A service provider continuously runs the server program

- lifetime of a server is infinite.
- lifetime of a client is finite.

Application Programming Interface

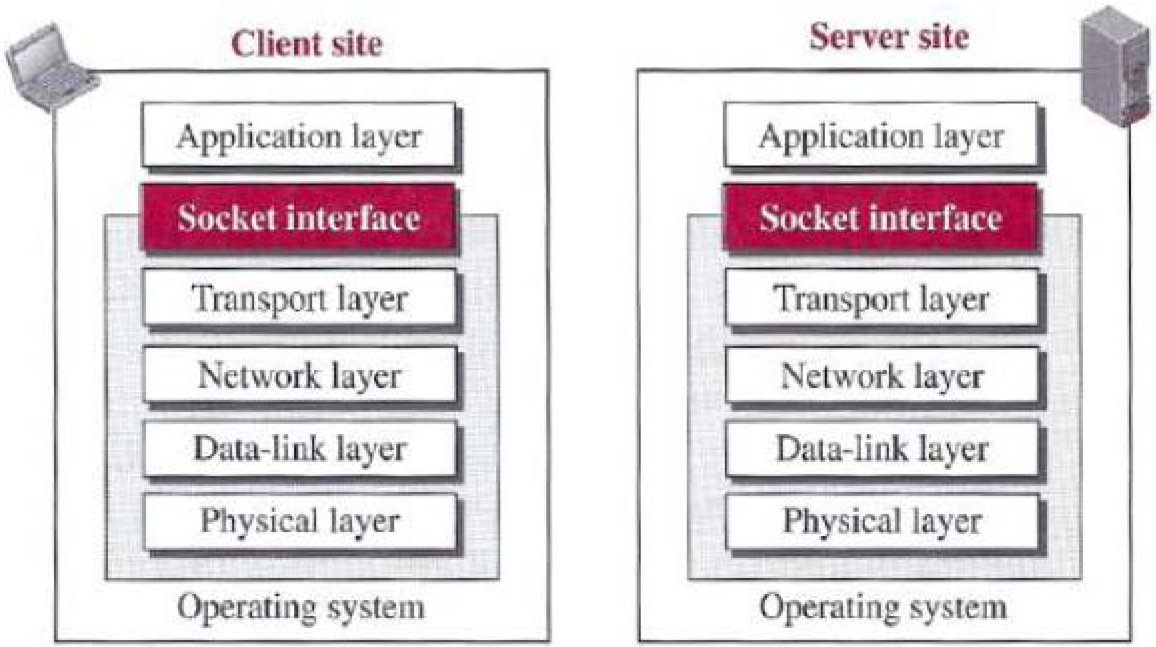
- How can a client process communicate with a server process?

- Let, Computer Programming
  - Uses a computer language to define what to do?
  - Has set of instructions for mathematical operations, string manipulation, input/output operation, etc.

- Application Programming
  - Set of instructions to talk with the lowest four layers (in OS)
  - instructs to open a connection, send and receive data, close the connection
  - Set of instruction of this kind is API

API

- Several APIs have been designed for communication
- Three most common APIs
  - Socket interface
  - Transport Layer Interface (TLI)
  - STREAM



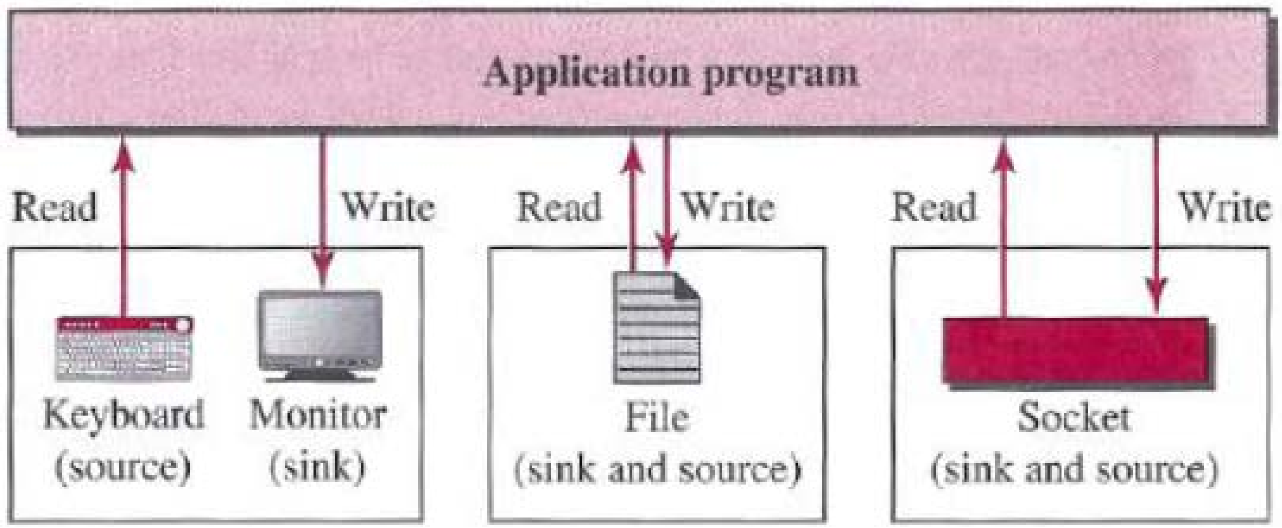
Socket Interface

- Socket interface started in the early 1980s at UC Berkeley as part of a UNIX environment.
- The socket interface is a set of instructions that provide communication between the application layer and the OS.

- The idea of sockets allows us to use the set of all instructions already designed in a programming language for other sources and sinks.
- For example,
  - in C, C++, or Java, we have several instructions that can read and write data to other sources and sinks;
  - a keyboard (a source), a monitor (a sink), or a file (source and sink).

- We are adding only new sources and sinks to the programming language without changing the way we send or receive data.

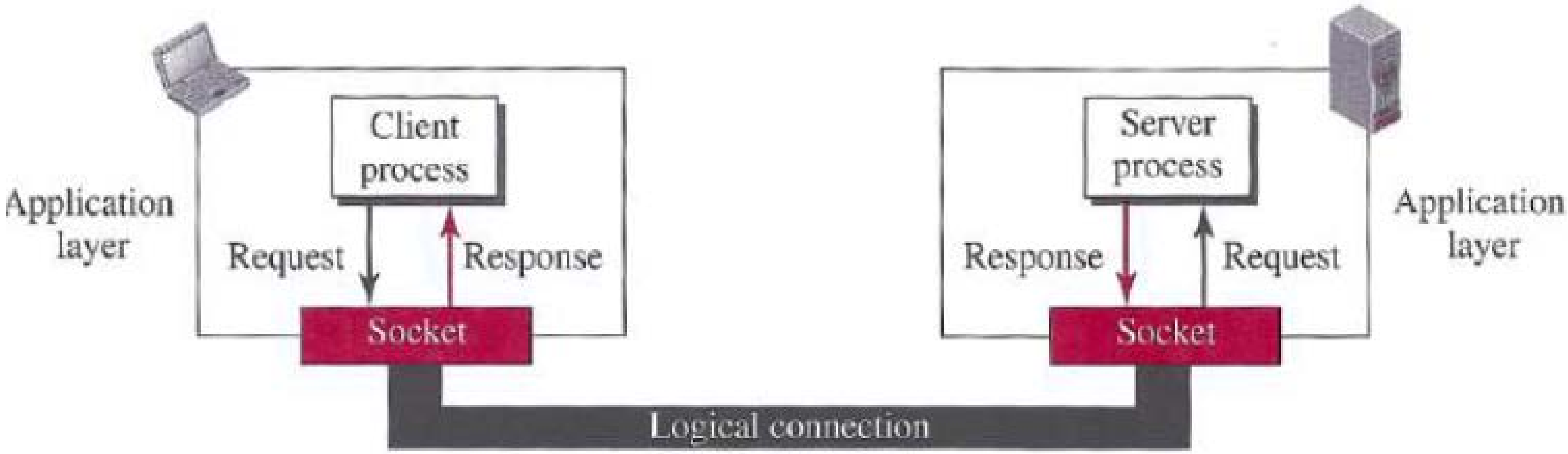
Cont…



- Socket is not a physical entity like them (files, keyboard, etc.); it is an abstraction

Cont…

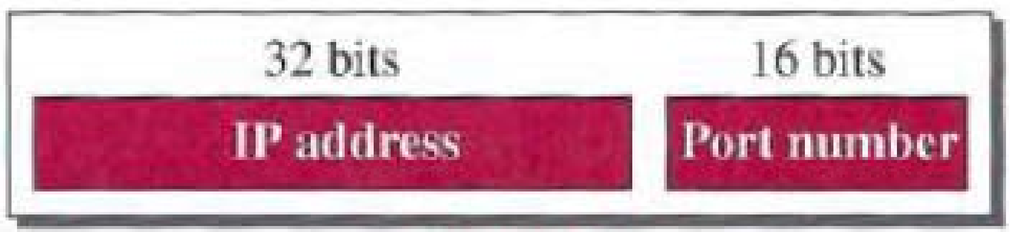
- Communication between a client process and a server process is communication between two sockets



- Need a pair of socket addresses for communication:
  - a local socket address and a remote socket address.

Cont…

- A socket address should first define the computer on which a client or a server is running.
  - a computer in the Internet is uniquely defined by its IP address
- Then, we need another identifier to define the specific client or server involved in the communication
  - an application program can be defined by a port number
- So, a socket address should be a combination of an IP address and a port number



Socket address

Finding Socket Addresses

- How can a client or a server find a pair of socket addresses for communication?

- In Server Site:
  - The server needs a local (server) and a remote (client) socket address for communication.
- In Client Site:
  - The client also needs a local (client) and a remote (server) socket address for communication.

Cont…

- In Server Site
  - *Local Socket Address (server)*: Provided by the OS; IP and Port number needs to be defined. For standard services, port numbers are well-known.
  - *Remote Socket Address (client)*: The server can find this socket address from the REQ packet when a client tries to connect to the server.

- In Client Site
  - *Local Socket Address (client)*: Provided by the OS; The port number is assigned to a client process each time the process needs to start the communication; the ephemeral port numbers are assigned to client
  - *Remote Socket Address (server)*: We know port-number of standard application, but don't know IP. We know only URL (e.g. www.gmail.com) , and DNS gives server socket address corresponding to URL.

Use Services of Transport Layer

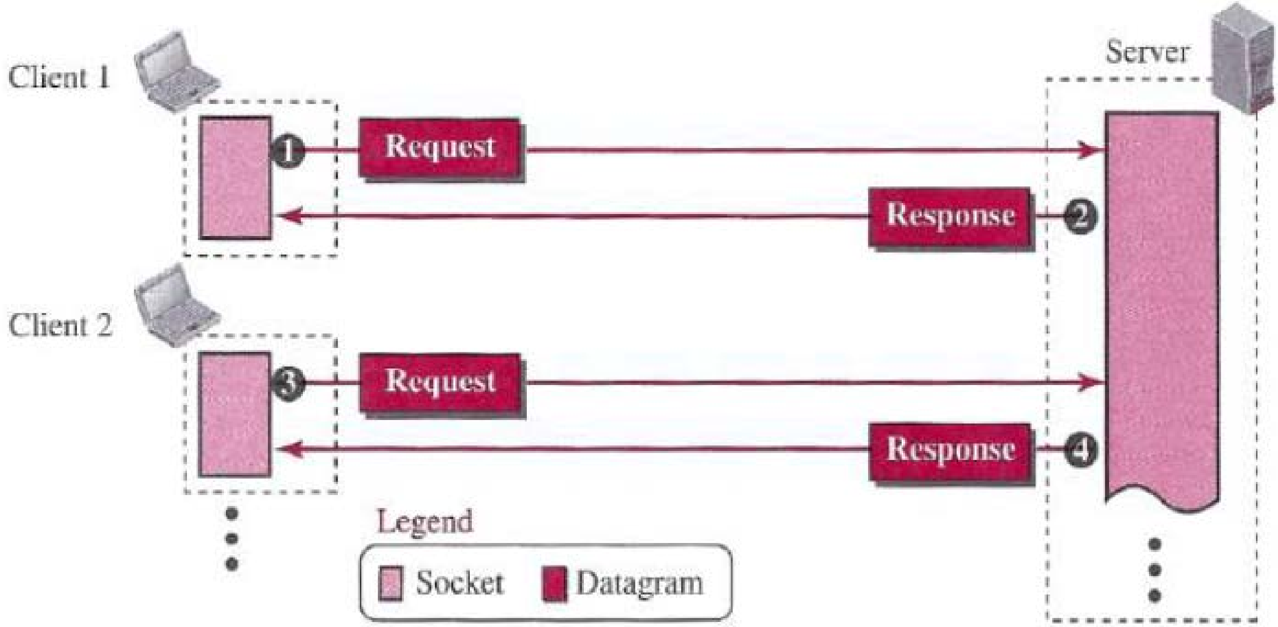
- The choice of the transport layer protocol seriously affects the capability of the application processes.

- use UDP
  - if it is sending small messages
  - if the simplicity and speed is more important for the application than reliability
- use TCP
  - if it needs to send long messages and require reliability

Iterative Communication Using UDP

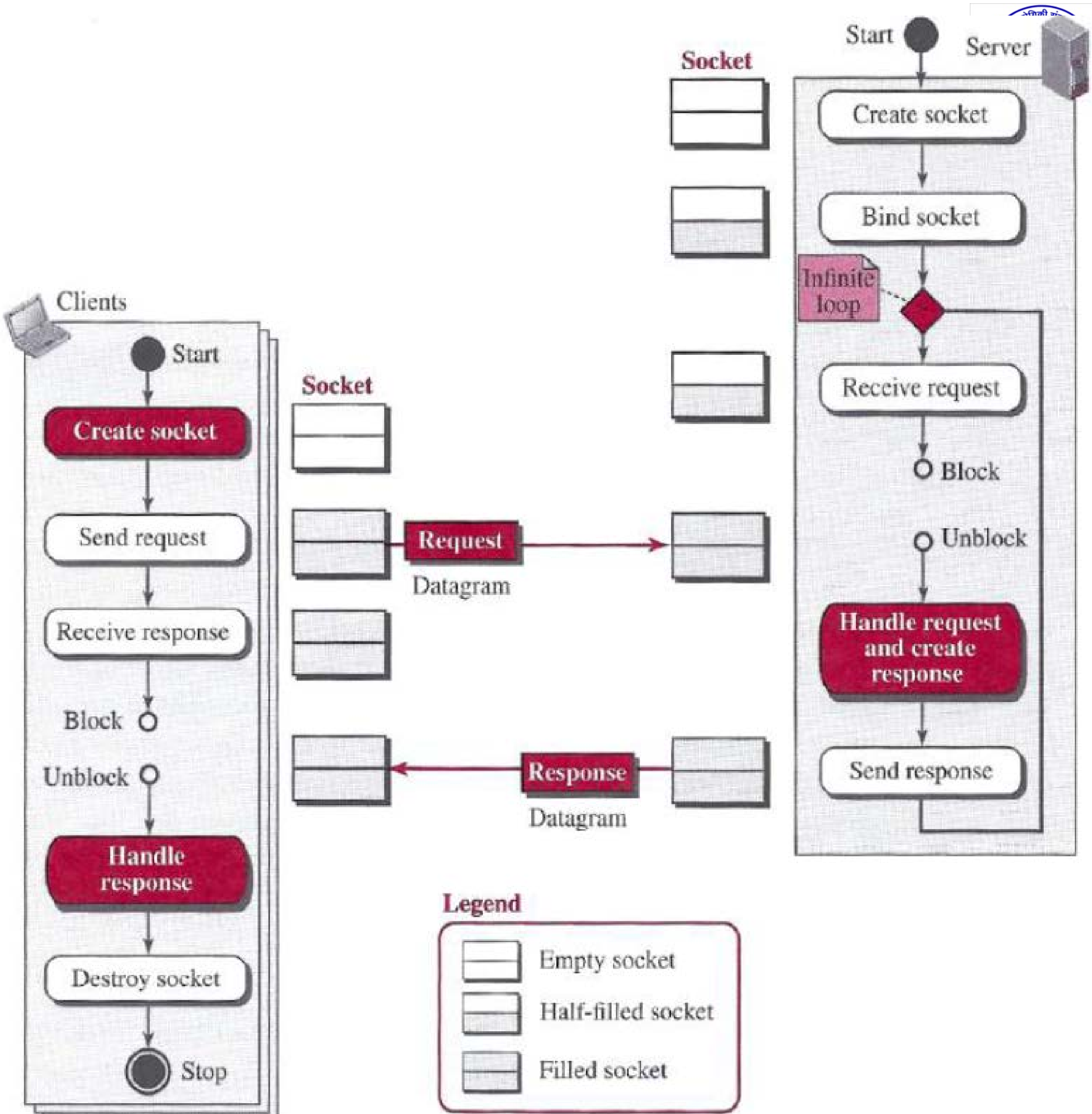
- several client programs can access the same server program at the same time
- the server program can be designed to respond
  - Iteratively (i.e. one by one)
  - Concurrently
- In UDP communication, the client and server use only one socket each
  - The socket created at the server site lasts forever;
  - the socket created at the client site is closed (destroyed) when the client process terminates.

Cont…



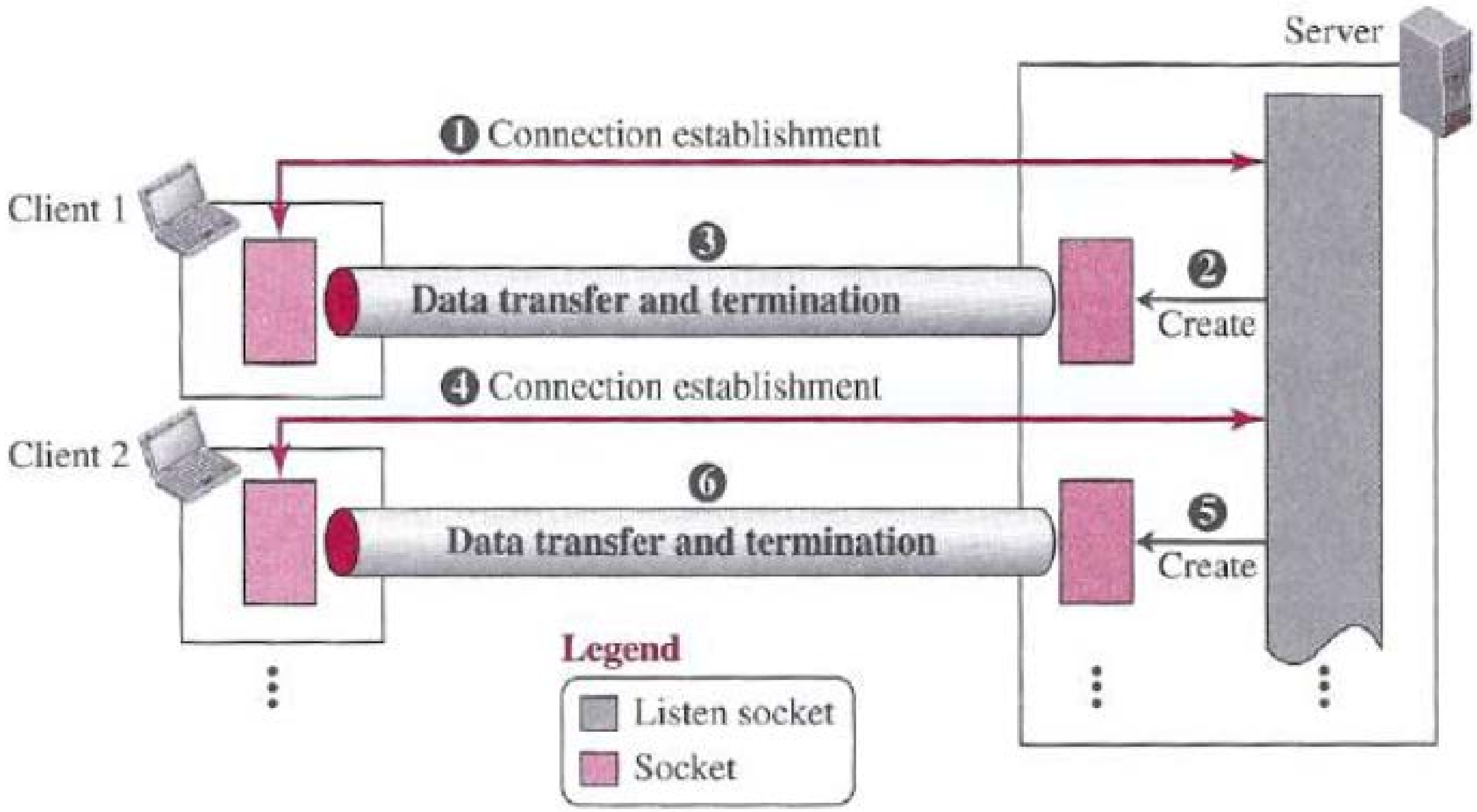
- Each client is served in each iteration of the loop in the server.
- Each client sends a single datagram and receives a single datagram.
- If a client wants to send two datagrams, it is considered as two clients for the server.

Flow

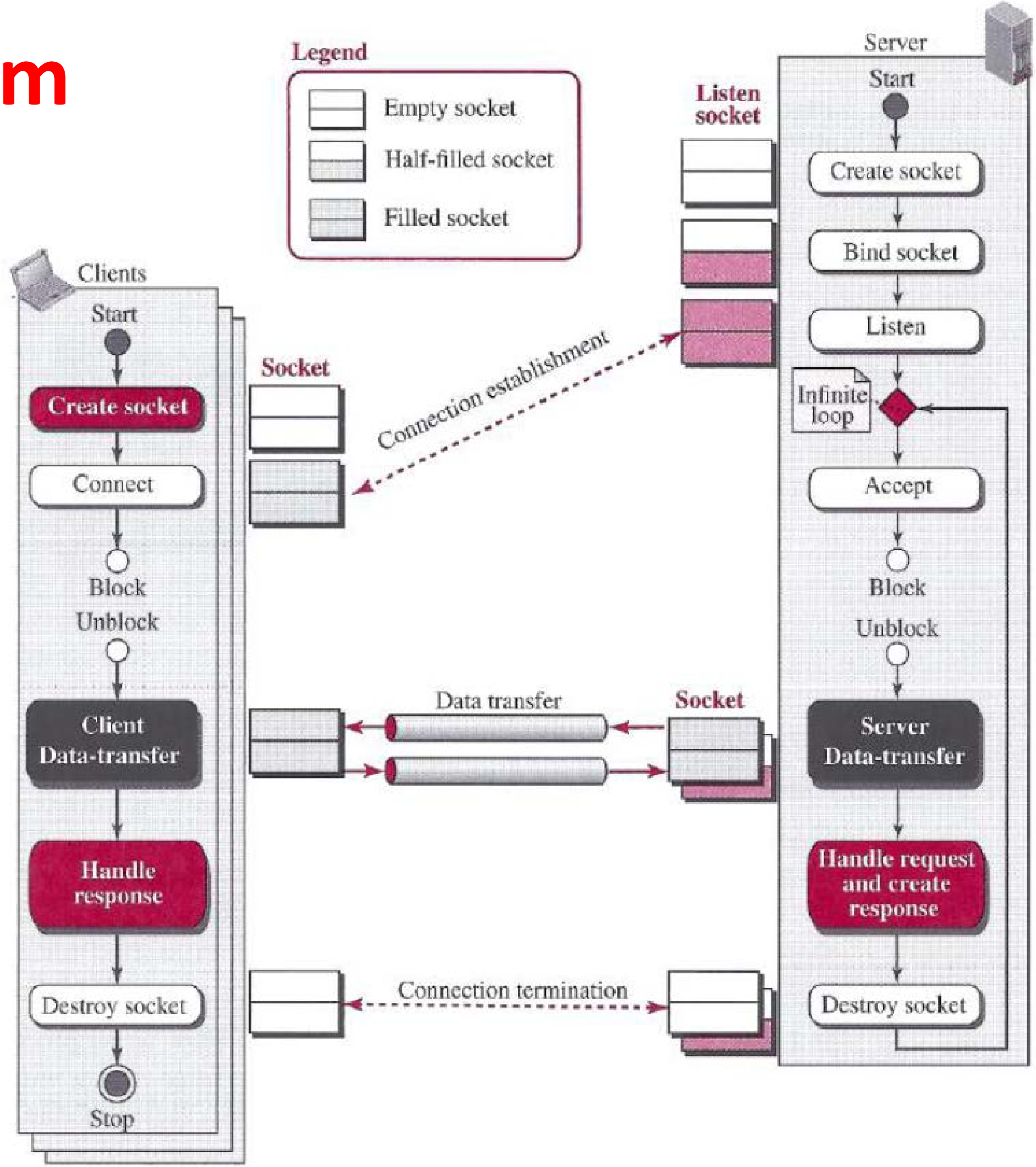

Clients
Start
Create socket
Send request
Receive response
Block
Unblock
Handle response
Destroy socket
Stop
Socket
Request
Datagram
Response
Datagram
Socket
Start
Server
Create socket
Bind socket
Infinite loop
Receive request
Block
Unblock
Handle request and create response
Send response
Legend
Empty socket
Half-filled socket
Filled socket

Iterative Communication Using TCP

- The TCP server uses two different sockets
  - one for connection establishment (*listen socket)*
  - the other for data transfer (*socket)*

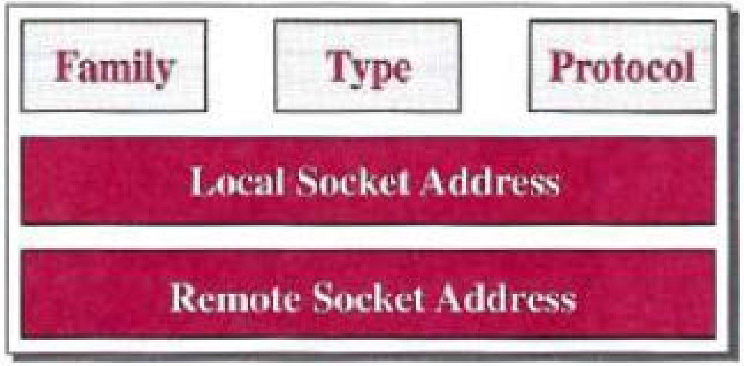


Flow Diagram



Iterative Socket Programming

- the role of a socket in communication
  - has no buffer to store data to be sent or received
  - is capable of neither sending nor receiving data
  - acts as a reference or a label
  - buffers and necessary variables are created inside OS
- The C language defines a socket as a structure.



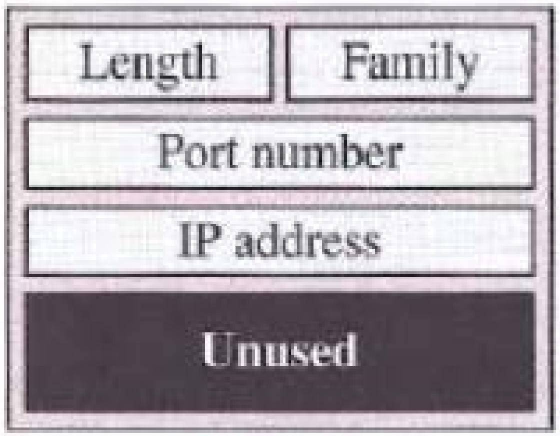
Socket

Cont…

- *Family:* defines the protocol family (PF). The common values are PF_INET (for current Internet),
- *Type:* defines four types of sockets
  - – SOCK_STREAM (for TCP),
  - – SOCK_DGRAM (for UDP),
  - – SOCK_SEQPACKET (for SCTP),
  - – SOCK_RAW (for directly use the IP)
- *Protocol:* defines the specific protocol in the family (e.g., 0 => TCP/IP)

Cont…

- *Local socket address*: defines the *local socket address*
  - *Length: size of the socket address*
  - *Family:* AF_INET for TCP/IP; *(AF: address family)*
  - *Port Number: it defines the process*
  - *IP Address: it* defines the host on which the process is running
  - *Unused: for future use*



Socket address

- *Remote socket address*: defines the remote socket address

Echo application using UDP

- The client program sends a short string of characters to the server;
- the server echoes back the same string to the client.

- "headerFiles.h" : need to use the definition of the socket and all procedures (functions) defined in the interface,

Socket

```
struct sockaddr_in {
    short               sin_family;      // e.g. AF_INET
    unsigned short      sin_port;        // e.g. htons(3490)
    struct in_addr      sin_addr;        // see struct in_addr, below. It is IP address.
    char                sin_zero[8];     // zero this if you want to
};
struct in_addr {
    unsigned long       s_addr;          // load with inet_aton()
};

struct   sockaddr_in   myaddr;
int s;
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);
inet_aton("63.161.169.137", &myaddr.sin_addr.s_addr);
//inet_aton::   convert from a struct in_addr to a string in dots-and-numbers

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct  sockaddr*)myaddr, sizeof(myaddr));
```

Family | Type | Protocol

Echo Server Program

- // **UDP** echo server program

```
#include "headerFiles.h"    // All header files required for socket programming

int main (void)
{  // Declare and define variables
int     s;                                   // Socket descriptor (reference)
int     len;                                 // Length of string to be echoed
char  buffer [256];                          // Data buffer
struct  sockaddr_in     servAddr;            // Server (local) socket address
struct  sockaddr_in     clntAddr;            // Client (remote) socket address
int      clntAddrLen;                        // Length of client socket address

// Build local (server) socket address
memset (&servAddr, 0, sizeof (servAddr));                // Allocate memory
servAddr.sin_family   = AF_INET;                         // Default Family field
servAddr.sin_port      = htons (SERVER_PORT)             // Default port number
servAddr.sin_addr.s_addr    = htonl (INADDR_ANY);        // Default IP address
```

Cont…

```
// Create socket
if ((s = socket (PF_INET, SOCK_DGRAM, 0) < 0);
{           perror ("Error: socket failed! ");
            exit (1);
}
// Bind socket to local address and port
If  (bind (s, (struct  sockaddr*) &servAddr, sizeof (servAddr)) < 0);
{           perror ("Error: bind failed!");
            exit (1);
}
for ( ; ; )  // Run forever
{           // Receive String
            len = recvfrom (s, buffer, sizeof (buffer), 0,
                              (struct sockaddr*)&clntAddr,  &clntAddrLen);
            // Send String
            sendto (s, buffer, len, 0, (struct sockaddr*)&clntAddr, sizeof(clntAddr));
} // End of for loop
} // End of echo server program
```

Echo Client Program

- //**UDP** echo client program

```
#include "headerFiles.h“

int main (int argc, char* argv[ ])     // Three arguments to be checked later
{ //Declare and define variables
int      s;                            // Socket descriptor
int      len;                          // Length of string to be echoed
char* servName;                        // Server name
int      servPort;                     // Server port
char*  string;                         // String to be echoed
char   buffer [256+ 1];                // Data buffer
struct  sockaddr_in   servAddr;   // Server socket address
// Check and set program arguments
if(argc !=3)
{ printf ("Error: three arguments are needed!");
  exit(1);
}
servName      =  argv[1];
servPort       =  atoi (argv[2]);
string          =  argv[3];
```

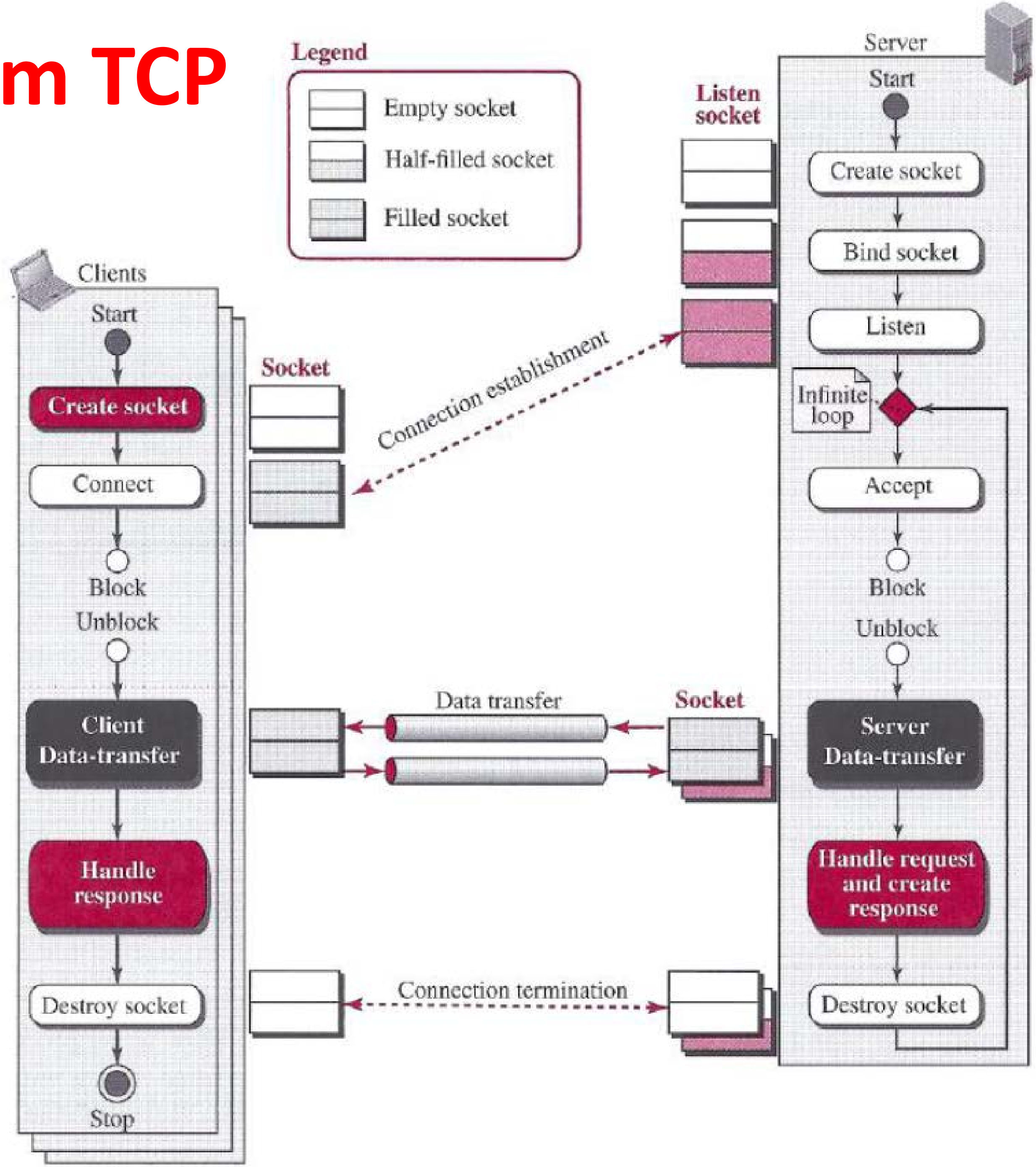
Cont…

```
// Build server socket address
memset (&servAddr, 0, sizeof (servAddr));
servAddr.sin_family = AF_INET;

//call DNS to find the server IP corresponding to server name
inet_pton (AF_INET, servName, &servAddr.sin_addr);
servAddr.sin_port = htons (servPort);

// Create socket
If ((s = socket (PF_INET, SOCK_DGRAM, 0) < 0);
{
perror ("Error: Socket failed!");
exit (1);
}
```

Cont…

```
//Send echo string
len = sendto (s, string, strlen (string), 0, (struct  sockaddr)&servAddr,
                    sizeof(servAddr));
//Receive echo string
recvfrom (s, buffer, len, 0, NULL, NULL);
// NULL: as we don’t need client socket address and length

//Print and verify echoed string
buffer [len] = '\0';
printf ("Echo string received: “);
fputs (buffer, stdout);
//Close the socket
close (s);
//Stop the program
exit (0);
} // End of echo client program
```

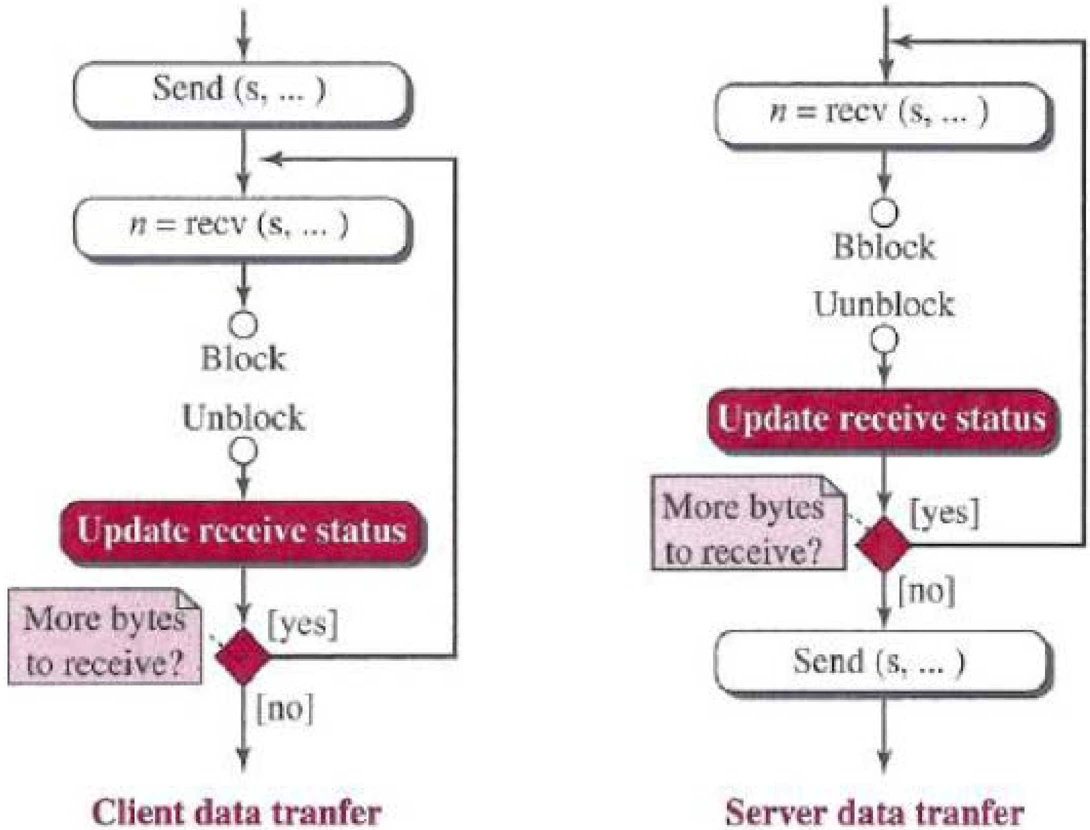
Flow Diagram TCP



Iterative Programming Using TCP

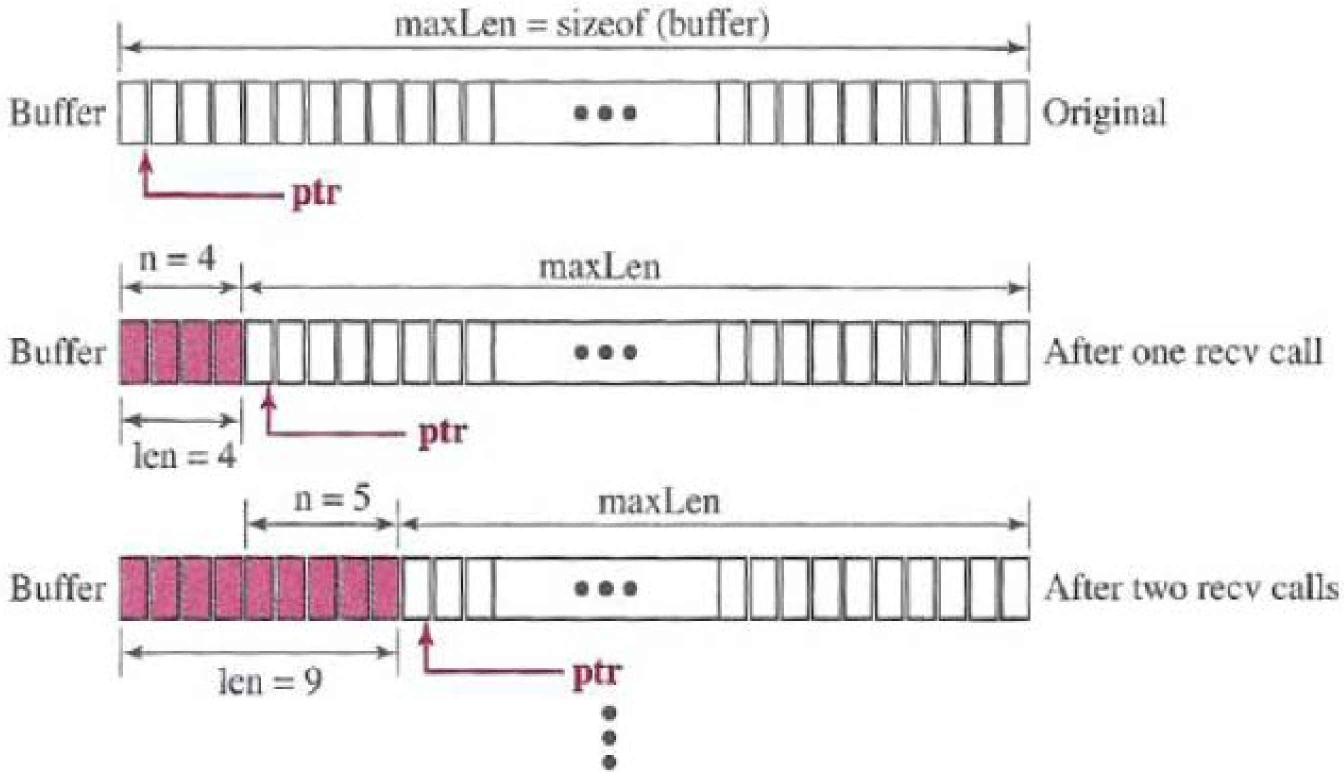
- Before sending or receiving data, a connection needs to be established between the client and the server.

Flow diagram for the client and server data-transfer boxes



Cont…

- We need to control
  - how many bytes of data we have received and
  - where the next chunk of data is stored.



Echo server program using TCP

```
// Echo server program
#include "headerFiles.h“

int main (void)
{
// Declare and define
int        ls;                         // Listen socket descriptor
int        s;                          // Socket descriptor (reference)
char       buffer [256];               // Data buffer
char*      ptr = buffer;               // Data buffer
int        len = 0;                    // Number of bytes to send or receive
int        maxLen = sizeof (buffer);   // Maximum number of bytes
int        n = 0;                      // Number of bytes for each recv call
int        waitSize = 16;              // Size of waiting clients
struct sockaddr _in   serverAddr;      // Server address
struct sockaddr_in    clientAddr;      // Client address
int        clntAddrLen;                // Length of client address
```

Cont…

```
// Create local (server) socket address
memset (&servAddr, 0, sizeof (servAddr));
servAddr.sin_family = AF_INET;
servAddr.sin_addr.s_addr = htonl(INADDR_ANY); // Default IP address
servAddr.sin_port = htons (SERV_PORT);        // Default port

// Create listen socket
if (ls = socket (PF_INET, SOCK_STREAM, 0) < 0)
{       perror ("Error: Listen socket failed!");
        exit (1);
}
// Bind listen socket to the local socket address
if (bind (ls, &servAddr, sizeof (servAddr)) < 0)
{       perror ("Error: binding failed!");
        exit (1);
}
```

Cont…

```
// Listen to connection requests
if (listen (ls, waitSize) < 0)
{         perror ("Error: listening failed!");
          exit (1);
}

// Handle the connection
for ( ; ; )        // Run forever
{
    // Accept connections from client
    if (s = accept (ls, &clntAddr, &clntAddrLen) < 0)
    {     perror ("Error: accepting failed!);
          exit (1);
    }
```

Cont…

// Data transfer section

```
while ((n = recv (s, ptr, maxLen, 0)) > 0)
{
      ptr += n ;         // Move pointer along the buffer
      maxLen -= n; // Adjust maximum number of bytes to receive
      len += n;          // Update number of bytes received
}
send (s, buffer, len, 0);  // Send back (echo) all bytes received
```

// Close the socket

```
close (s);

} // End of for loop
} // End of echo server program
```

Echo client program using TCP

```
// TCP echo client program
#include "headerFiles.h“

int main (int argc, char* argv[ ])
{  // Declare and define
int         s;                   // Socket descriptor
int         n;                   // Number of bytes in each recv call
char*       servName;            // Server name
int         servPort;            // Server port number
char*       string;              // String to be echoed
int         len;                 // Length of string to be echoed
char        buffer [256 + 1];    // Buffer
char*       ptr = buffer;        // Pointer to move along the buffer
struct      sockaddr_in   serverAddr;     // Server socket address

// Check and set arguments
if (argc != 3)
{       printf ("Error: three arguments are needed!");
        exit (1);
}
```

Cont…

```
servName = arg [1];
servPort = atoi (arg [2]);
string = arg [3];

II Create remote (server) socket address
memset (&servAddr, 0, sizeof(servAddr));

serveAddr.sin_family = AF _INET;
inet_pton (AF_INET, servName,&serveAddr.sin_addr);     // Server IP
serveAddr.sin_port = htons (servPort);                 // Server port

//Create socket
if ((s = socket (PF_INET, SOCK_STREAM, 0) < 0);
{   perror ("Error: socket creation failed!");
    exit (1);
}
```

Cont…

```
// Connect to the server
if (connect (sd, (struct sockaddr*)&servAddr, sizeof(servAddr)) < 0)
{     perror ("Error: connection failed!");
      exit (1);
}

// Data transfer section
send (s, string, strlen(string), 0);

while ((n = recv (s, ptr, maxLen, 0)) > 0)
{
    ptr + = n;              // Move pointer along the buffer
    maxLen - = n;           // Adjust the maximum number of
    len += n;               // Update the length of string received
}
```

Cont…

```
// Print and verify the echoed string
buffer [len] = '\0';
printf ("Echoed string received: ");
fputs (buffer, stdout);

// Close socket
close (s);

// Stop program
exit (0);
} // End of echo client program
```

Thanks!