Network Socket Programming - 3

BUPT/QMUL

2010-10-26

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Electronic Engineering

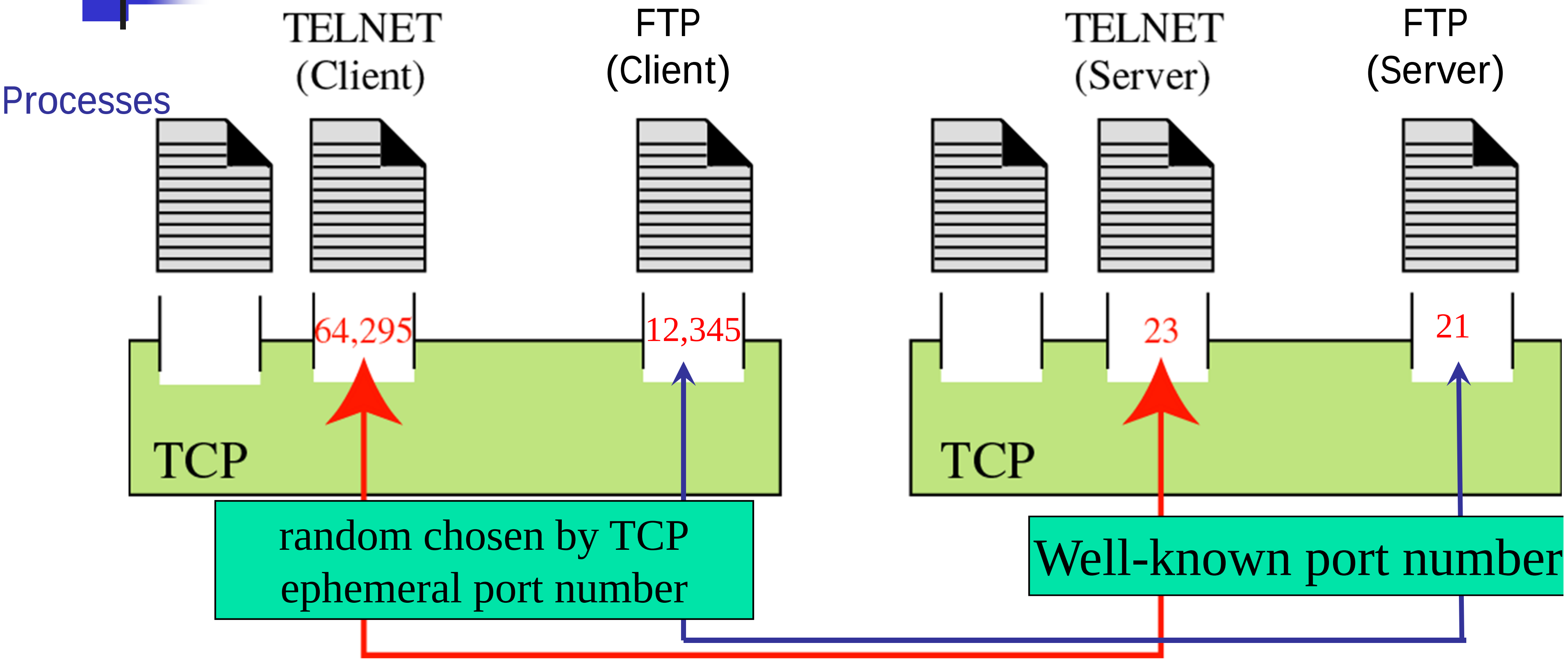
Agenda

- Basic concepts in NP
- Introduction to IP & TCP/UDP
- *Introduction to Sockets*

Introduction to Sockets

- *Reviews of some helpful points*
- *Sockets interface*
- Major system calls
- Sample programs

Connection and Port Number



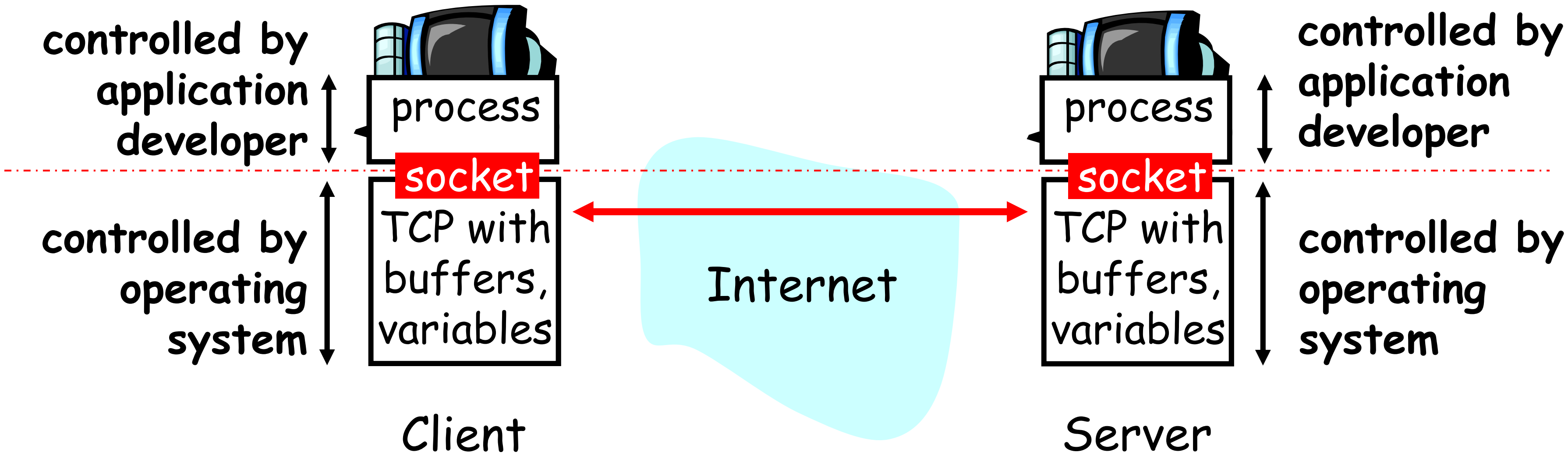
A connection is identified by (Source IP address, Source Port Number, Destination IP address, Destination Port Number)

Understanding Socket

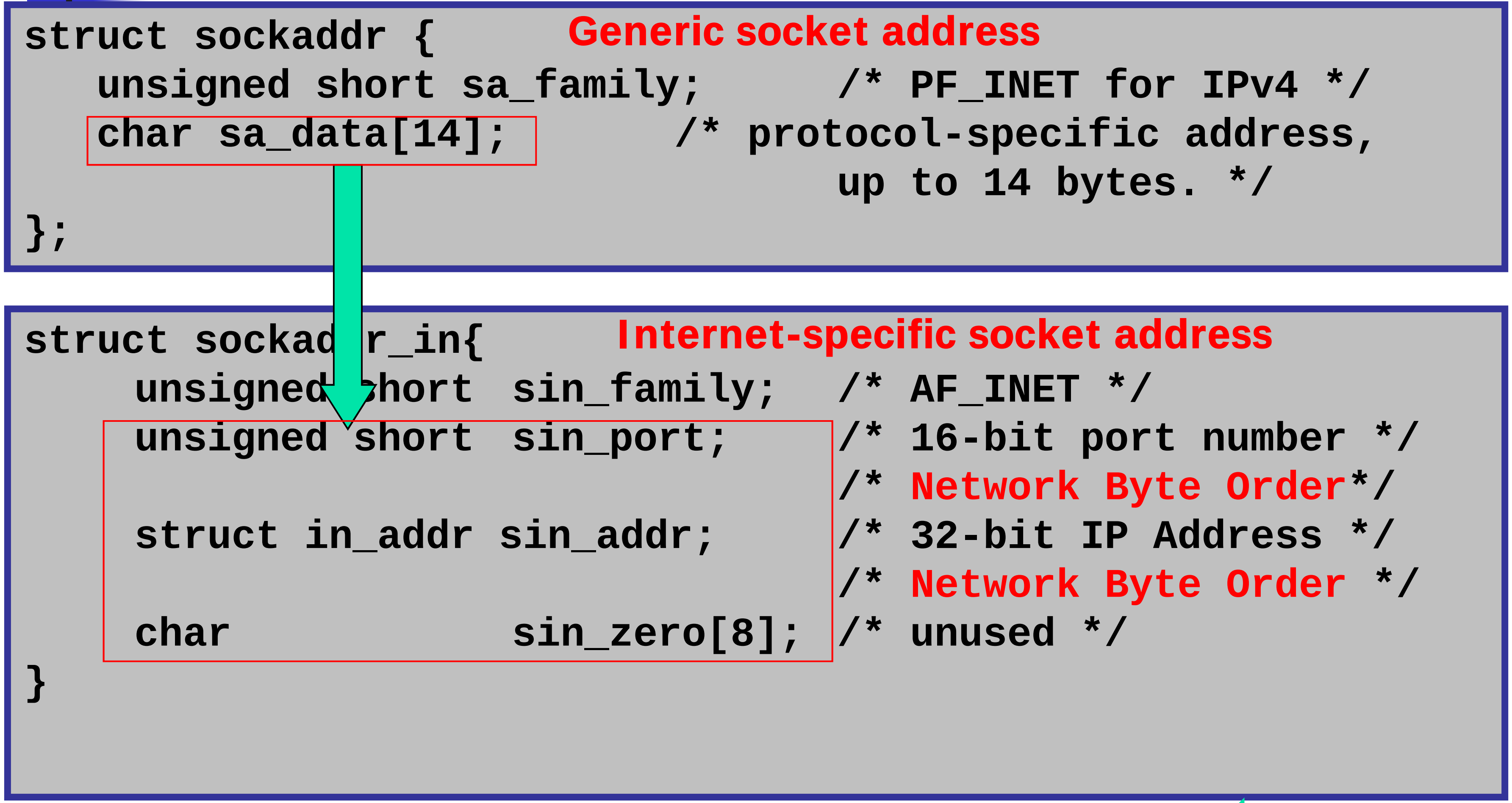
- An extension to OS's I/O system, enabling communication between processes and machines
- A host-local, application-created/owned, OS-controlled interface (a "door") into which application process can both send and receive messages to/from another (remote or local) application process
- A socket can be treated the same as a standard file descriptor except that
  - It is created with the socket()
  - Additional calls are needed to connect and activate it
  - recv() and send() are also used as counterparts to read() and write()

Socket Programming using TCP

- Socket: a door between application process and end-end-transport protocol (UDP or TCP)
- TCP service: reliable transfer of bytes from one process to another


controlled by application developer
process
socket
TCP with buffers, variables
controlled by operating system
Client
Internet
process
socket
TCP with buffers, variables
controlled by application developer
controlled by operating system
Server

Socket Address



Generic socket address

```
struct sockaddr {
   unsigned short sa_family;      /* PF_INET for IPv4 */
   char sa_data[14];          /* protocol-specific address,
                                 up to 14 bytes. */
};
```

Internet-specific socket address

```
struct sockaddr_in{
     unsigned short  sin_family;  /* AF_INET */
     unsigned short  sin_port;    /* 16-bit port number */
                                  /* Network Byte Order*/
     struct in_addr sin_addr;     /* 32-bit IP Address */
                                  /* Network Byte Order */
     char            sin_zero[8]; /* unused */
}
```

Socket Programming: Telephone Analogy

- A telephone call over a “telephony network” works as follows:
  - Both parties have a telephone installed.
  - A phone number is assigned to each telephone.
  - Turn on ringer to listen for a caller.
  - Caller lifts telephone and dials a number.
  - Telephone rings and the receiver of the call picks it up.
  - Both Parties talk and exchange data.
  - After conversation is over they hang up the phone.

Dissecting the Analogy

- A network application works as follows:
  - An endpoint (telephone) for communication is created on both ends.
  - An address (phone no) is assigned to both ends to distinguish them from the rest of the network.
  - One of the endpoints (caller) initiate a connection to the other.
  - The other endpoint(receiver) waits for the communication to start.
  - Once a connection has been made, data is exchanged (talk).
  - Once data has been exchanged the endpoints are closed (hang up).

In the world of sockets……

- Socket() – Endpoint for communication
- Bind() - Assign a unique telephone number.
- Listen() – Wait for a caller.
- Connect() - Dial a number.
- Accept() – Receive a call.
- Send(), Recv() – Talk.
- Close() – Hang up.

Introduction to Sockets
Part III: major system calls

System Calls

- Socket operation
- Byte order operation
- Address formats conversion
- Socket option
- Name and address operation

System Calls – Socket Operation

- socket()
  - returns a socket descriptor
- bind()
  - What address I am on / what port to attach to
- connect()
  - Connect to a remote host
- listen()
  - Waiting for someone to connect to my port
- accept()
  - Get a file descriptor for a incoming connection
- send() and recv()
  - Send and receive data over a connection
- sendto() and recvfrom()
  - Send and receive data without connection
- close() and shutdown()
  - **Close a connection Two way / One way**
- readn(), writen(), readline()
  - Read / Write a particular number of bytes

System Calls – Byte Order Operation

- htonl()
  - Convert long int from host byte order to network byte order
- htons()
  - Convert short int from host byte order to network byte order
- ntohl()
  - Convert long int from network byte order to host byte order
- ntohs()
  - Convert short int from network byte order to host byte order

System Calls – Address Formats Conversion

- **inet_aton()**
  - converts an IP address in numbers-and-dots notation (ASCII string) into unsigned long in network byte order
- inet_addr()
  - converts an IP address in numbers-and-dots notation (ASCII string) into unsigned long in network byte order
- **inet_ntoa()**
  - Mapping a 32-bit integer (an IP address in network byte order) to an ASCII string in dotted decimal format
- **inet_pton()**
  - Similar to inet_aton() but working with IPv4 and IPv6
- **inet_ntop()**
  - Similar to inet_ntoa() but working with IPv4 and IPv6

System Calls – Socket Option

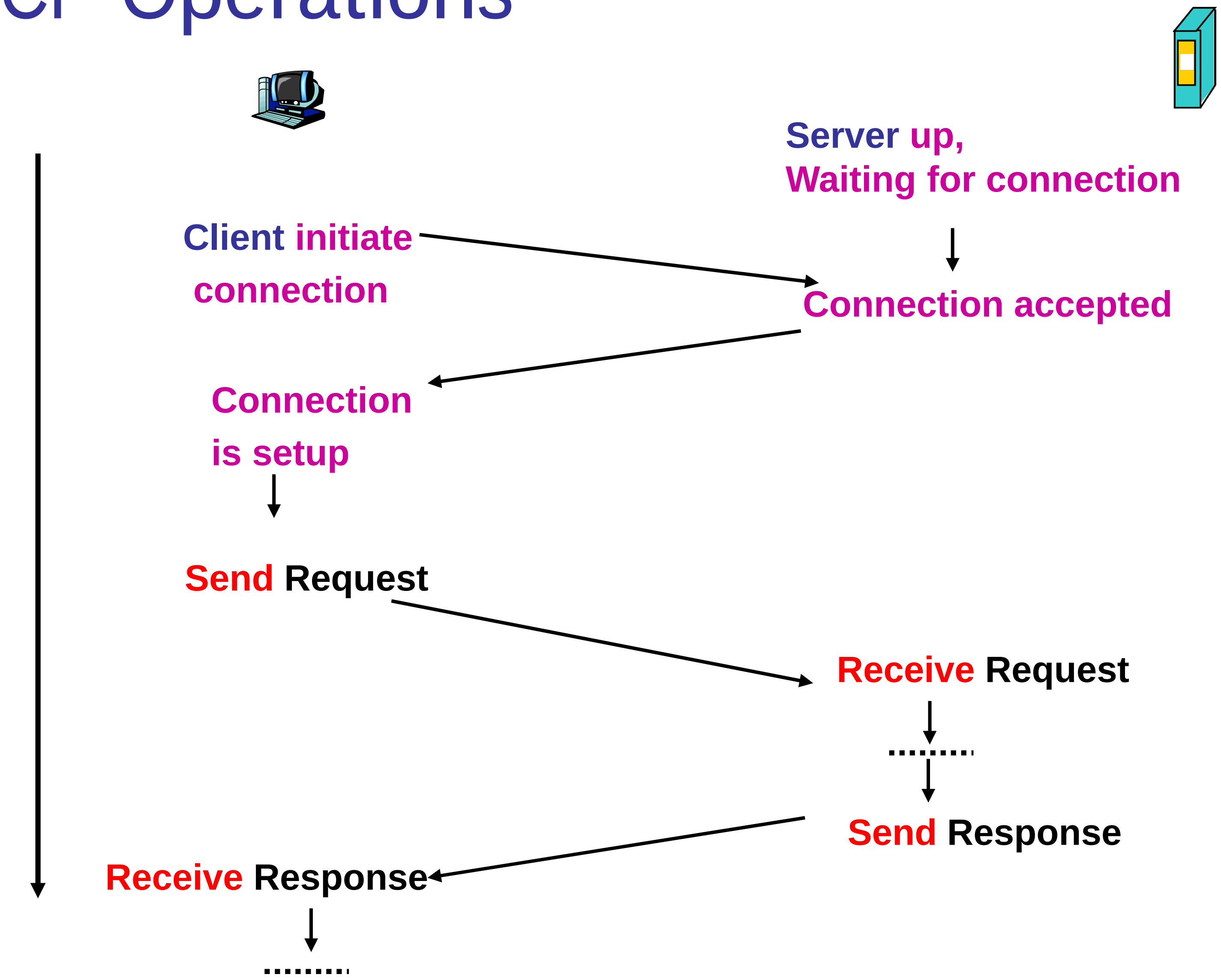
- getsockopt()
  - Allow an application to require information about the socket
- setsockopt()
  - Allow an application to set a socket option

  - eg.
    - get/set sending/receiving buffer size of a socket

System Calls – Name and Address Operation

- **gethostbyname()**
  - retrieving host entries from DNS and the query key is a DNS domain name
- **gethostbyaddr()**
  - retrieving host entries from DNS and the query key is an IP address
- gethostname()
  - Obtaining the name of a host
- getservbyname()
  - Mapping a named service onto a port number
- getservbyaddr()
  - Obtaining an entry from the services database given the port number assigned to it

Process of Socket Operation: TCP Operations



Server up,
Waiting for connection

Client initiate connection

Connection accepted

Connection is setup

Send Request

Receive Request

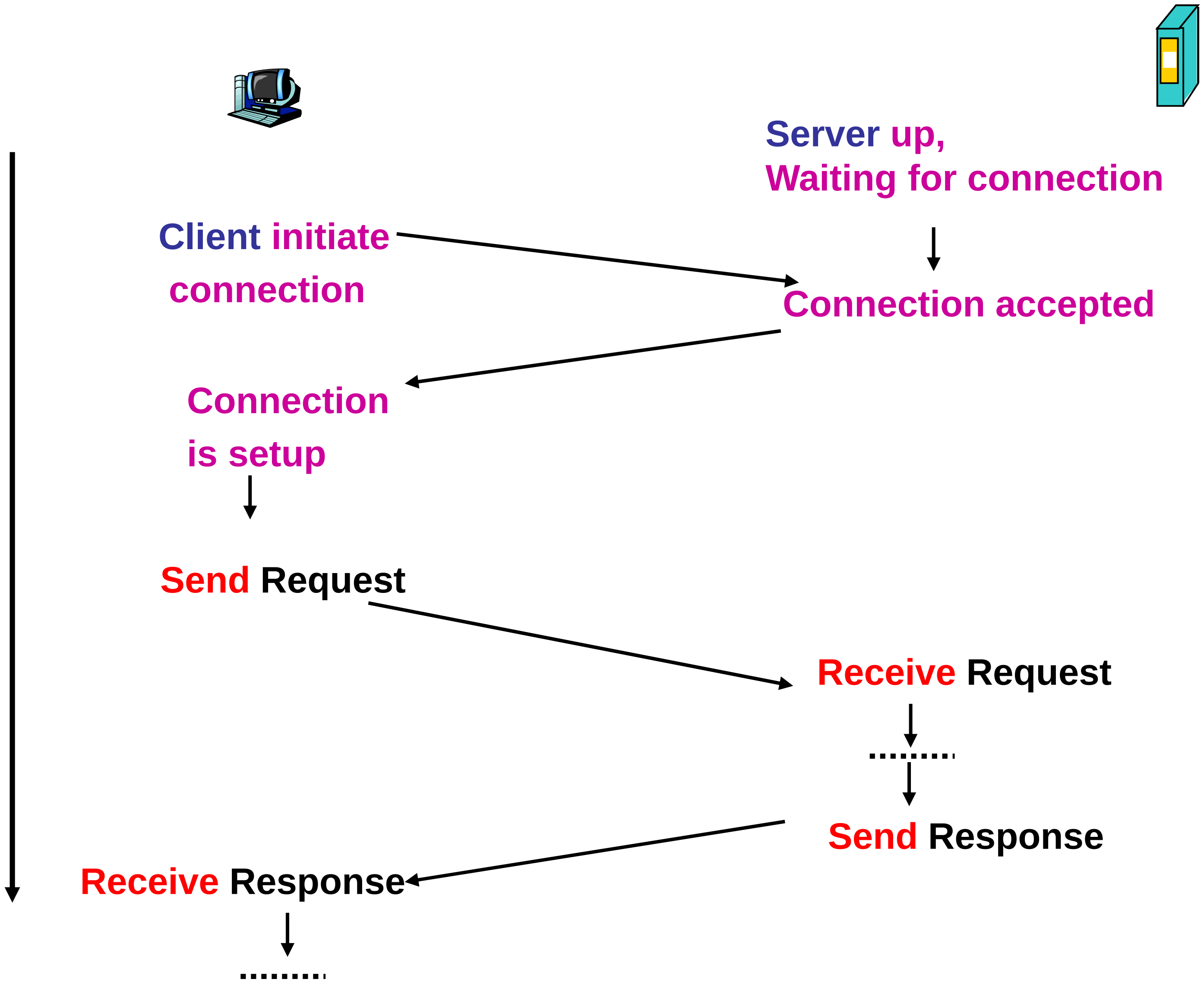
....

Send Response

Receive Response

....

Map to TCP socket programming



Server up,
Waiting for connection

Client initiate connection

Connection accepted

Connection is setup

Send Request

Receive Request

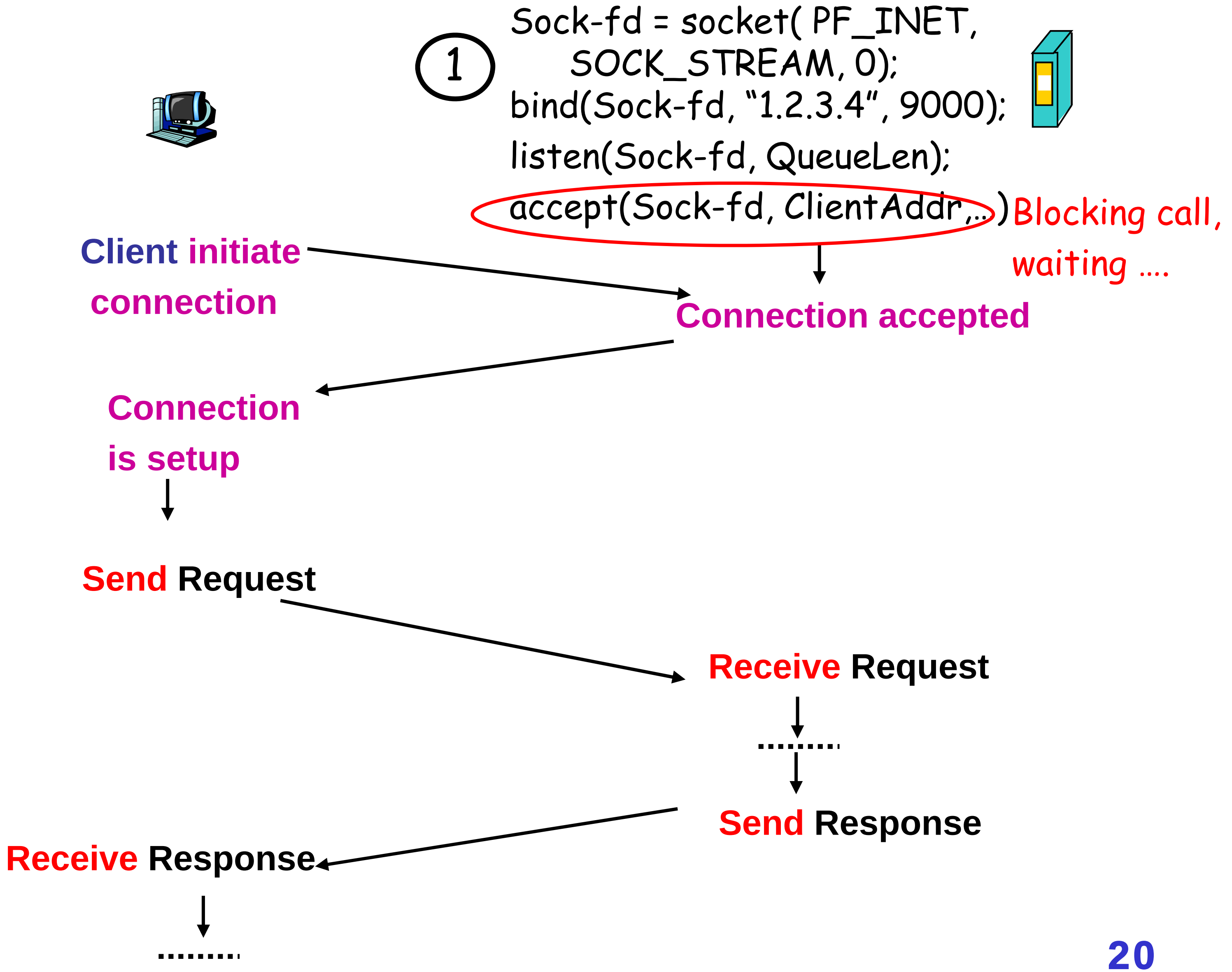
....

Send Response

Receive Response

....

Map to TCP socket programming



1

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,..)
```

Blocking call, waiting ….

Client initiate connection

Connection accepted

Connection is setup

Send Request

Receive Request

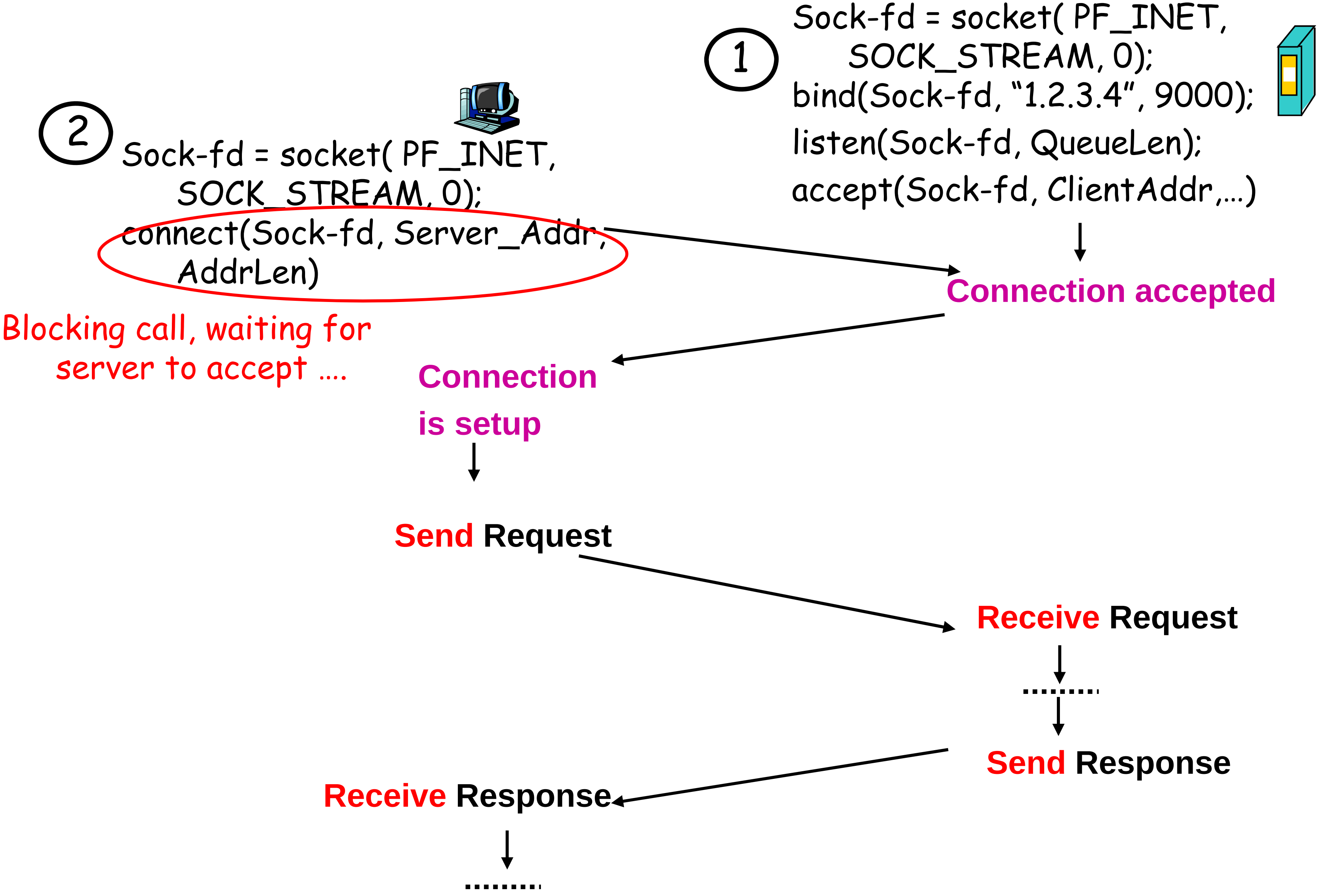
……….

Send Response

Receive Response

………

Map to TCP socket programming



1

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)
```

2

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
     AddrLen)
```

Blocking call, waiting for
server to accept ….

Connection accepted

Connection
is setup

Send Request

Receive Request

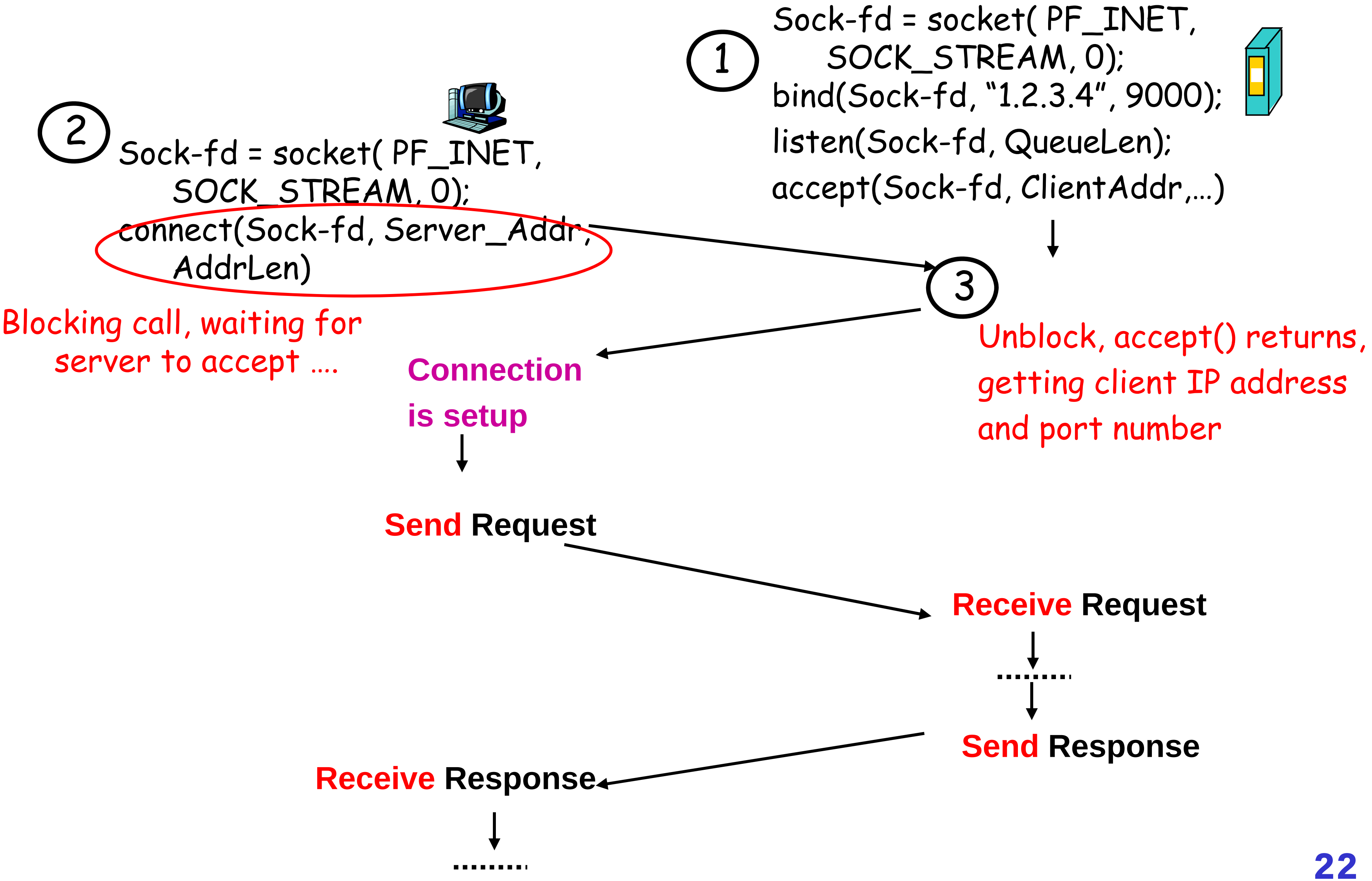
……….

Send Response

Receive Response

………

Map to TCP socket programming



1

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
bind(Sock-fd, "1.2.3.4", 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,...)
```

2

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
     AddrLen)
```

Blocking call, waiting for server to accept ….

3

Unblock, accept() returns, getting client IP address and port number

Connection is setup

Send Request

Receive Request

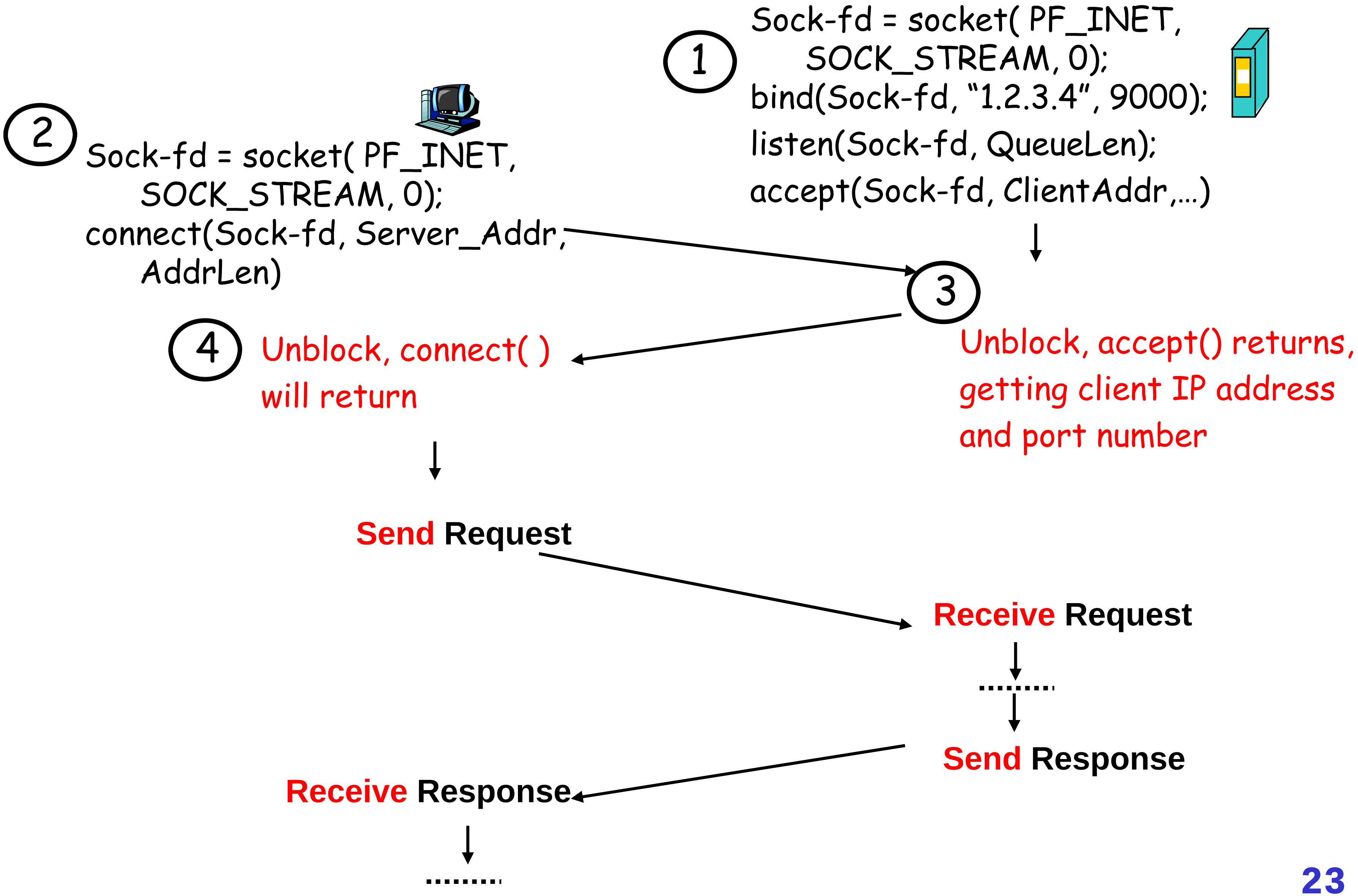
………

Send Response

Receive Response

………

Map to TCP socket programming



1 Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)

2 Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
AddrLen)

3 Unblock, accept() returns, getting client IP address and port number

4 Unblock, connect( ) will return

Send Request

Receive Request

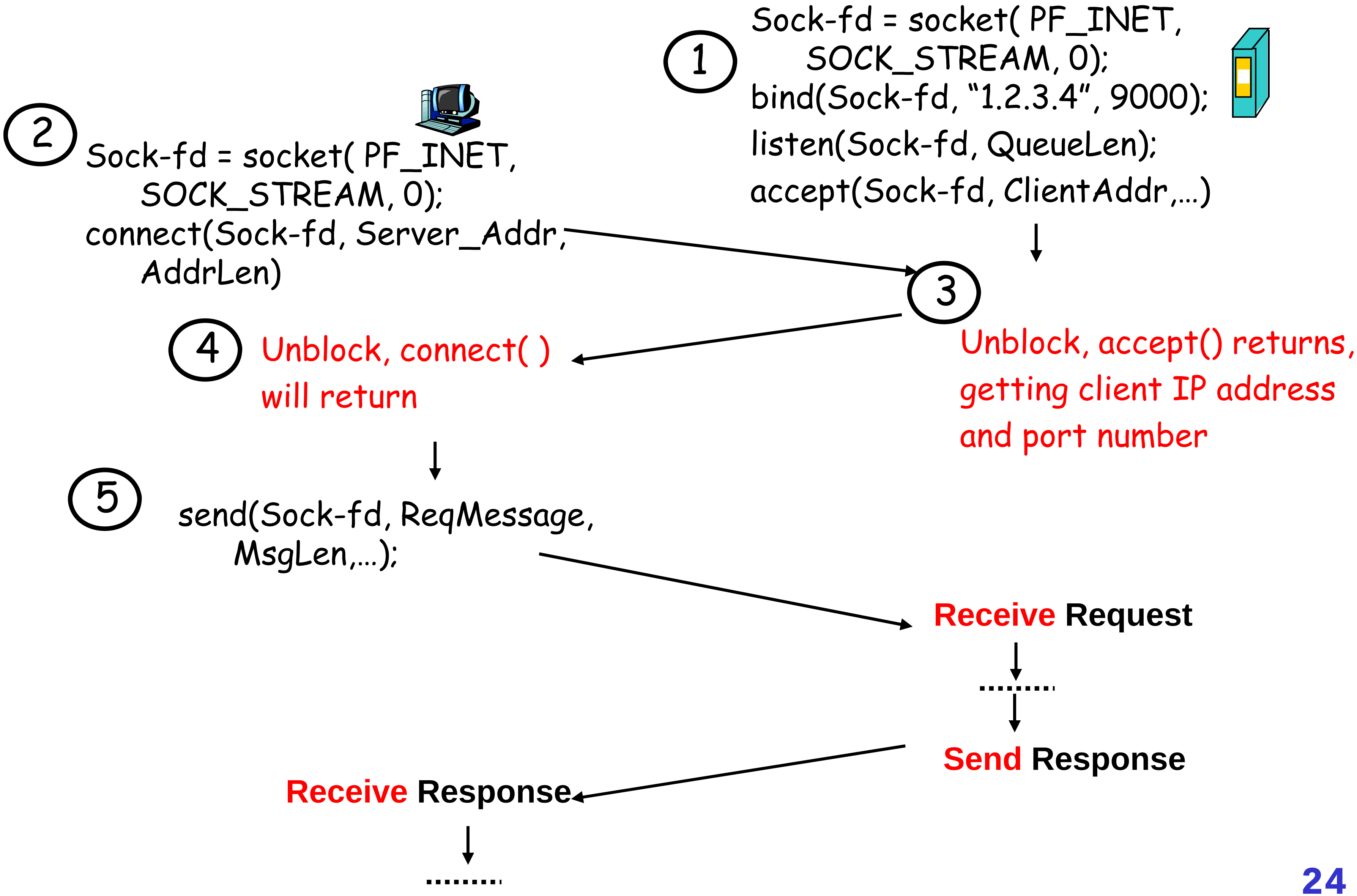
………

Send Response

Receive Response

………

Map to TCP socket programming



1

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)
```

2

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
     AddrLen)
```

3

Unblock, accept() returns, getting client IP address and port number

4

Unblock, connect( ) will return

5

```
send(Sock-fd, ReqMessage,
     MsgLen,…);
```

Receive Request

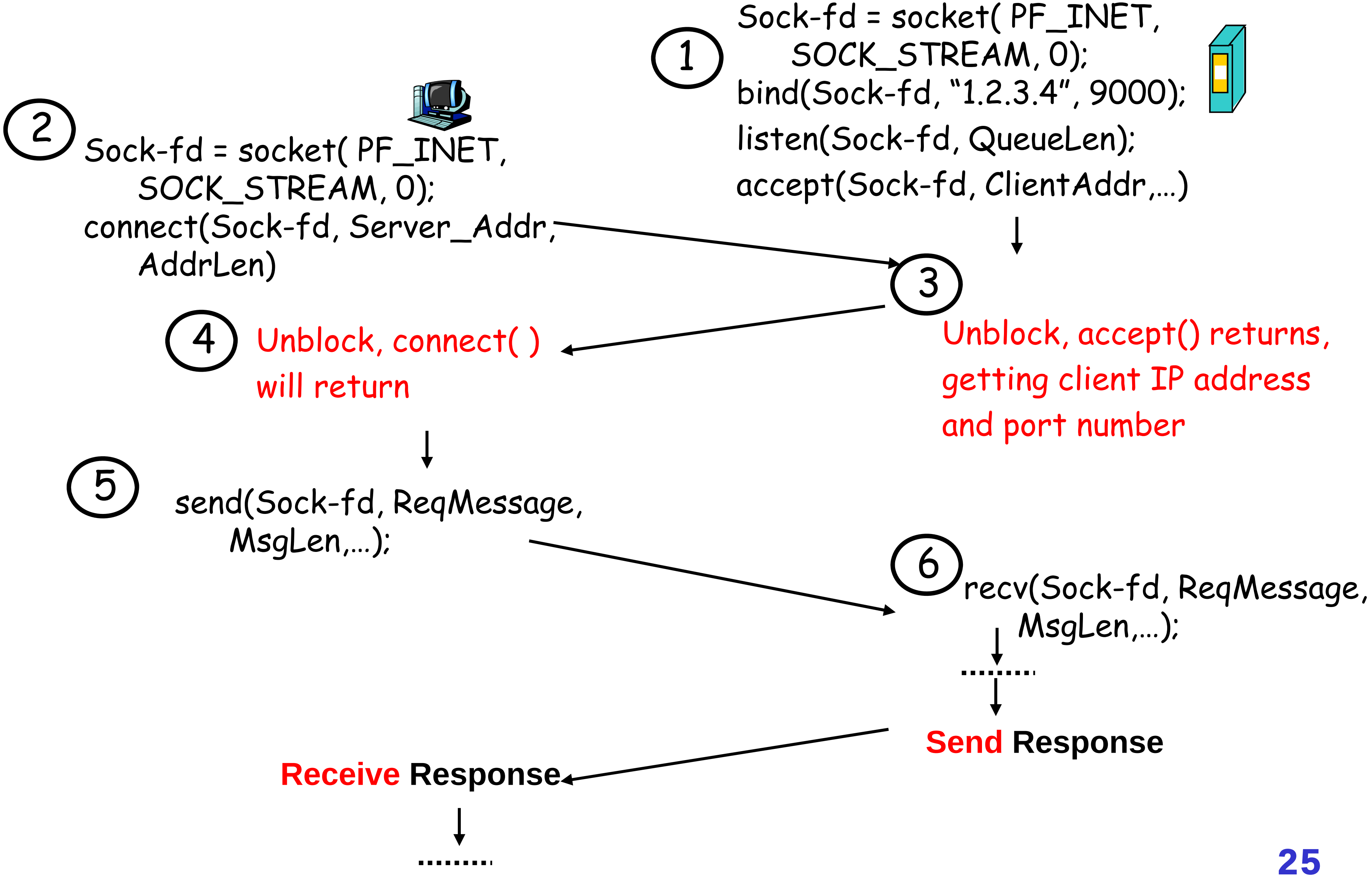
………

Send Response

Receive Response

………

Map to TCP socket programming



1 Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)

2 Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
AddrLen)

3 Unblock, accept() returns,
getting client IP address
and port number

4 Unblock, connect( )
will return

5 send(Sock-fd, ReqMessage,
MsgLen,…);

6 recv(Sock-fd, ReqMessage,
MsgLen,…);

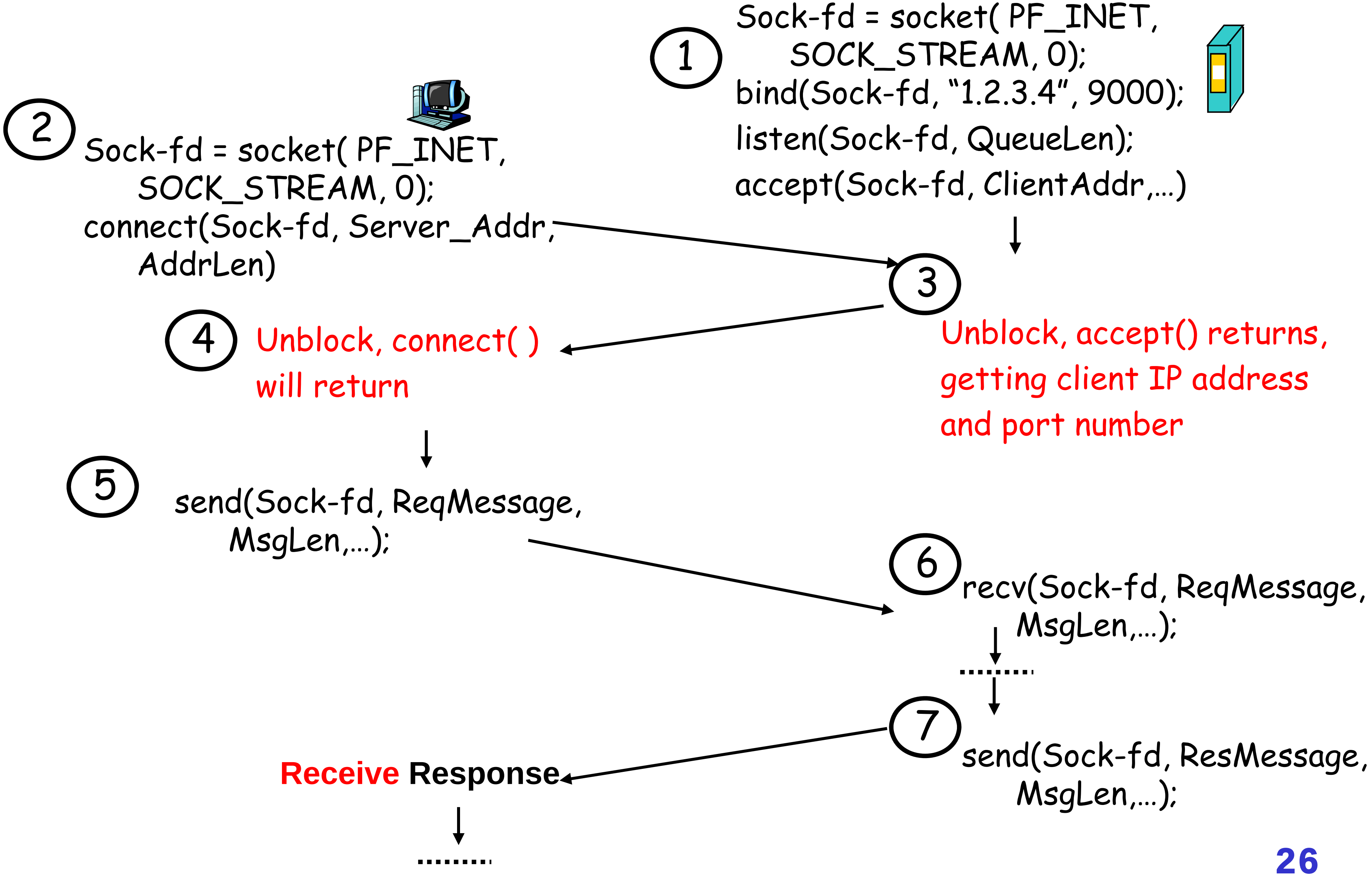
………

Send Response

Receive Response

………

Map to TCP socket programming



1 Sock-fd = socket(PF_INET,
    SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)

2 Sock-fd = socket(PF_INET,
    SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
    AddrLen)

3 Unblock, accept() returns,
getting client IP address
and port number

4 Unblock, connect( )
will return

5 send(Sock-fd, ReqMessage,
    MsgLen,…);

6 recv(Sock-fd, ReqMessage,
    MsgLen,…);

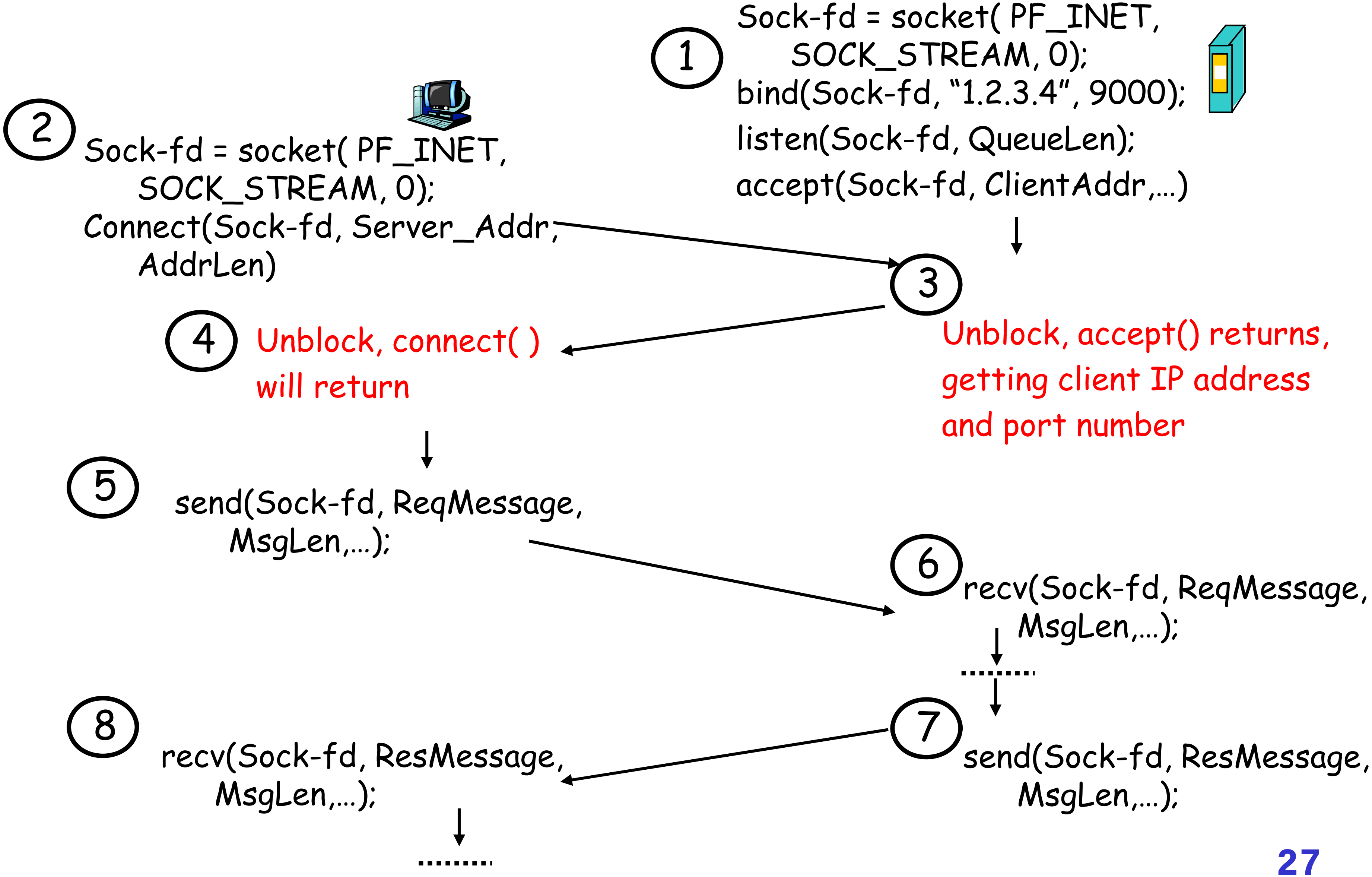
………

7 send(Sock-fd, ResMessage,
    MsgLen,…);

Receive **Response**

………

Map to TCP socket programming



1

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,...)
```

2

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
Connect(Sock-fd, Server_Addr,
     AddrLen)
```

3

Unblock, accept() returns, getting client IP address and port number

4

Unblock, connect( ) will return

5

```
send(Sock-fd, ReqMessage,
     MsgLen,...);
```

6

```
recv(Sock-fd, ReqMessage,
     MsgLen,...);
```

....

7

```
send(Sock-fd, ResMessage,
     MsgLen,...);
```

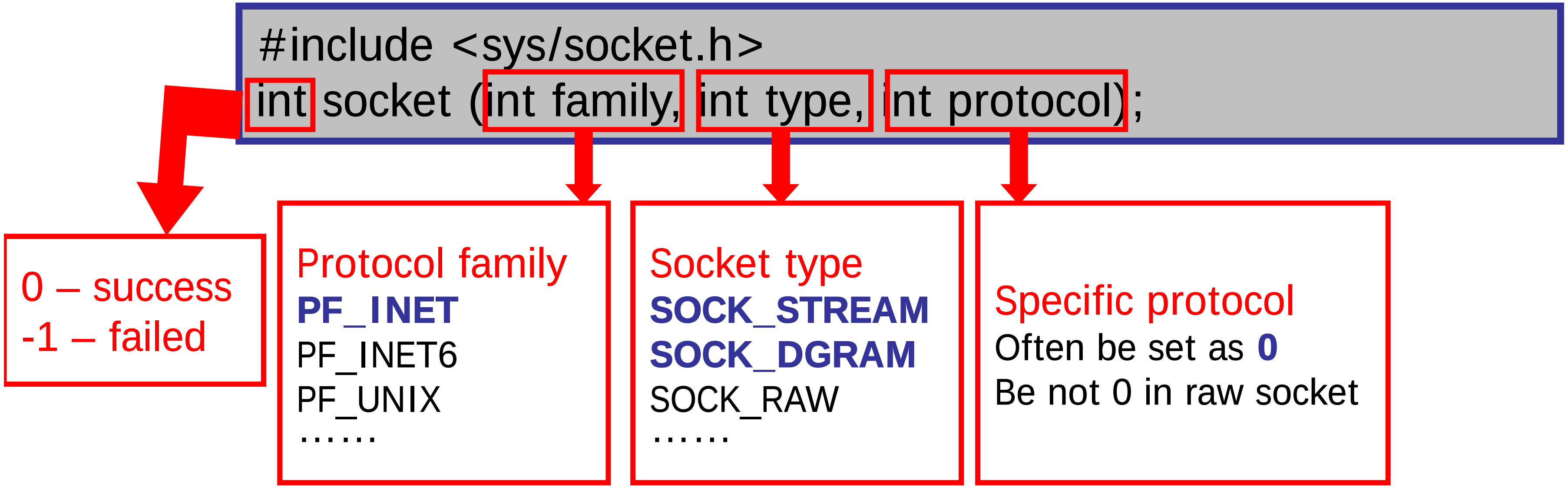
8

```
recv(Sock-fd, ResMessage,
     MsgLen,...);
```

....

System Calls – socket()

- An application calls *socket()* to create a new socket that can be used for network communication
- The call returns a descriptor for the newly created socket



```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
```

0 – success
-1 – failed

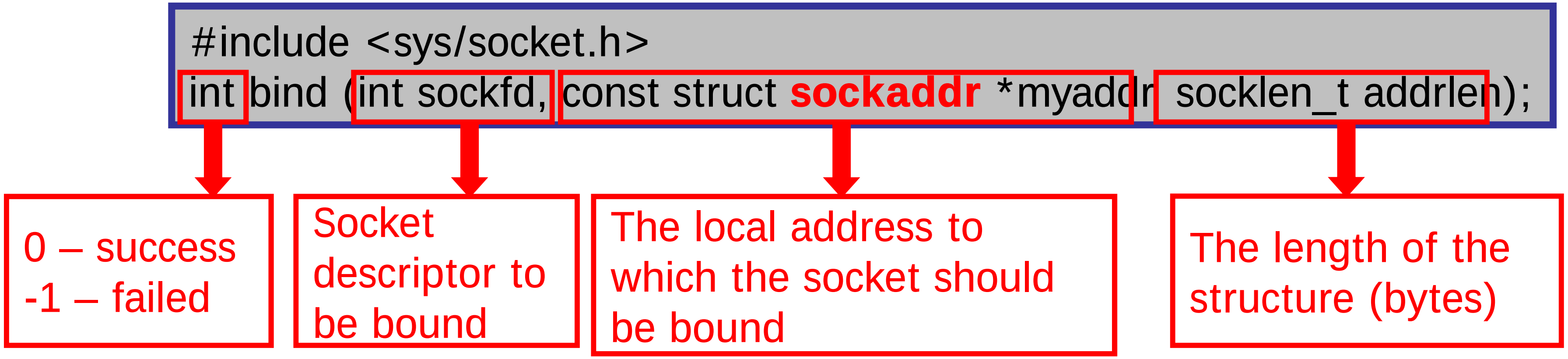
Protocol family
PF_INET
PF_INET6
PF_UNIX
……

Socket type
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
……

Specific protocol
Often be set as **0**
Be not 0 in raw socket

System Calls – bind()

- An application calls bind() to specify the local endpoint address (a **local IP address and protocol port number**) for a socket
- For TCP/IP, the endpoint address uses the ***sockaddr_in*** structure
- **Servers** use *bind* to specify the well-known port at which they will await connections



```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

0 – success
-1 – failed

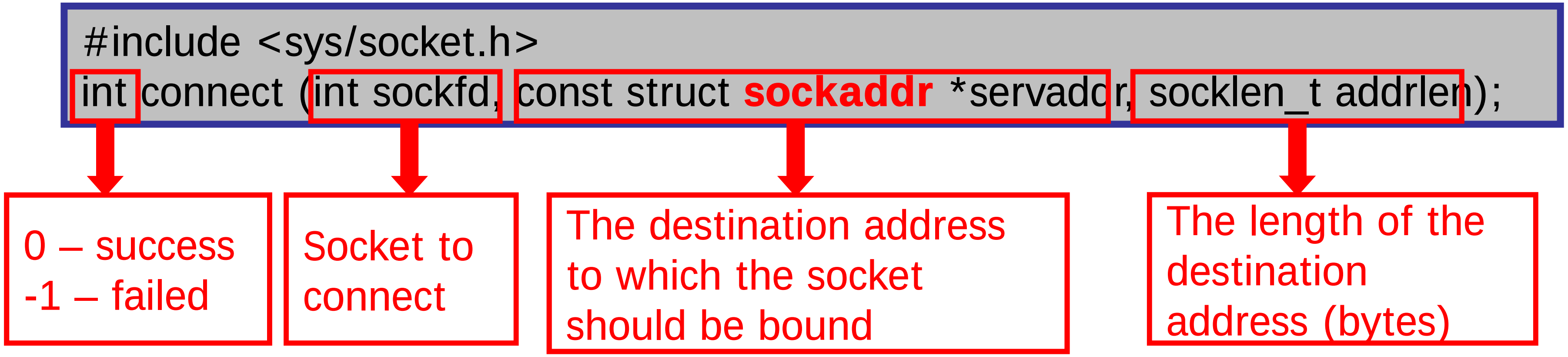
Socket descriptor to be bound

The local address to which the socket should be bound

The length of the structure (bytes)

System Calls – connect()

- After creating a socket, a **client** calls *connect()* to establish an active connection to a remote server



```
#include <sys/socket.h>
int connect (int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

0 – success
-1 – failed

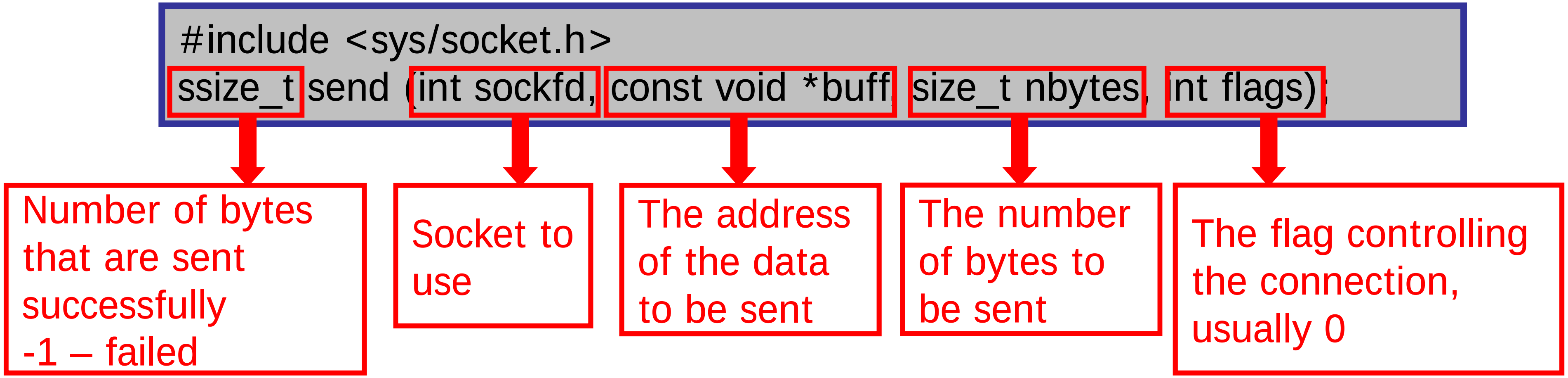
Socket to connect

The destination address to which the socket should be bound

The length of the destination address (bytes)

System Calls – send()

- Both clients (to transmit request) and servers (to transmit replies) used *send()* to transfer data across a TCP connection
- The application passes the descriptor of a socket to which the data should be sent, the address of the data to be sent, and the length of the data
- Usually, send copies outgoing data into buffers in the OS kernel



```
#include <sys/socket.h>
ssize_t send (int sockfd, const void *buff, size_t nbytes, int flags);
```

Number of bytes that are sent successfully
-1 – failed

Socket to use

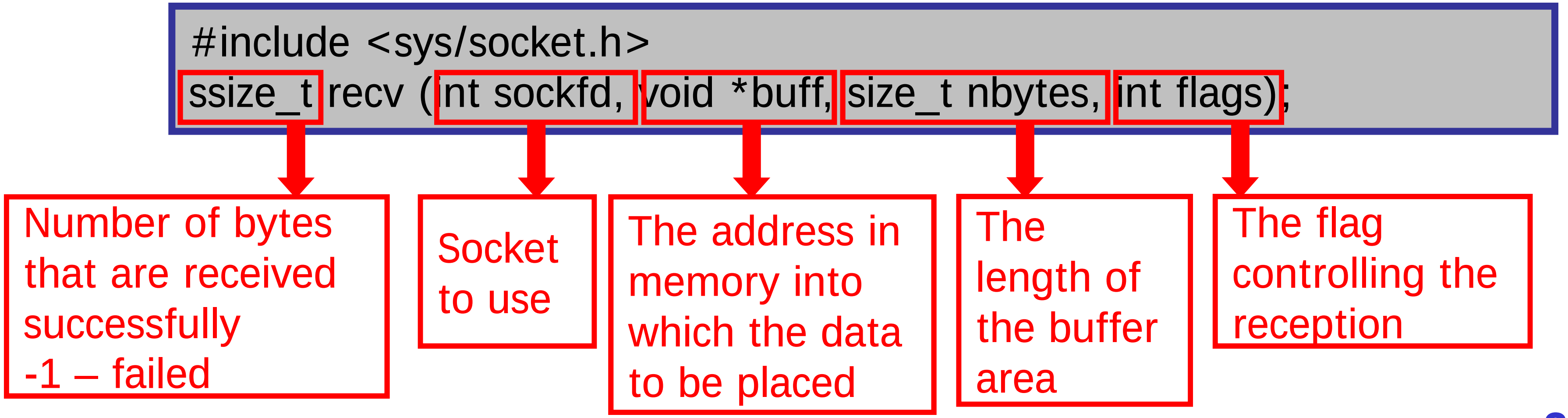
The address of the data to be sent

The number of bytes to be sent

The flag controlling the connection, usually 0

System Calls – recv()

- Both clients (to receive a reply) and servers (to receive a request) use *recv* to receive data from a TCP connection
- Clients and server can also use *recv* to receive messages from sockets that use UDP
- If the buffer cannot hold an incoming user datagram, *recv* fills the buffer and discards the remainder

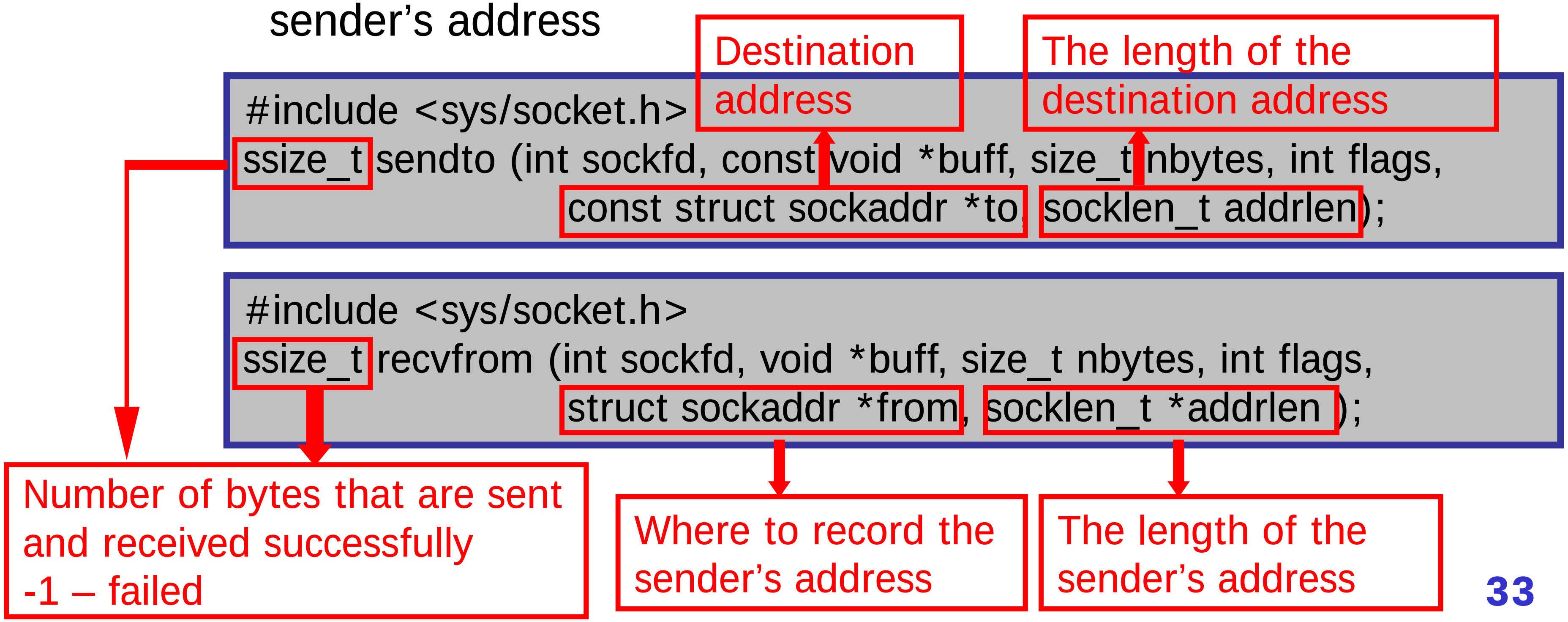


```
#include <sys/socket.h>
ssize_t recv (int sockfd, void *buff, size_t nbytes, int flags);
```

Number of bytes that are received successfully
-1 – failed
Socket to use
The address in memory into which the data to be placed
The length of the buffer area
The flag controlling the reception

System Calls – sendto() & recvfrom()

- Allow the caller to send or receive a message over a **UDP** connection
- sendto() require the caller to specify a destination
- recvfrom() uses an argument to specify where to record the sender's address



```
#include <sys/socket.h>
ssize_t sendto (int sockfd, const void *buff, size_t nbytes, int flags,
                const struct sockaddr *to, socklen_t addrlen);
```

```
#include <sys/socket.h>
ssize_t recvfrom (int sockfd, void *buff, size_t nbytes, int flags,
                struct sockaddr *from, socklen_t *addrlen );
```

Destination address

The length of the destination address

Number of bytes that are sent and received successfully
-1 – failed

Where to record the sender's address

The length of the sender's address

Using Read and Write with sockets

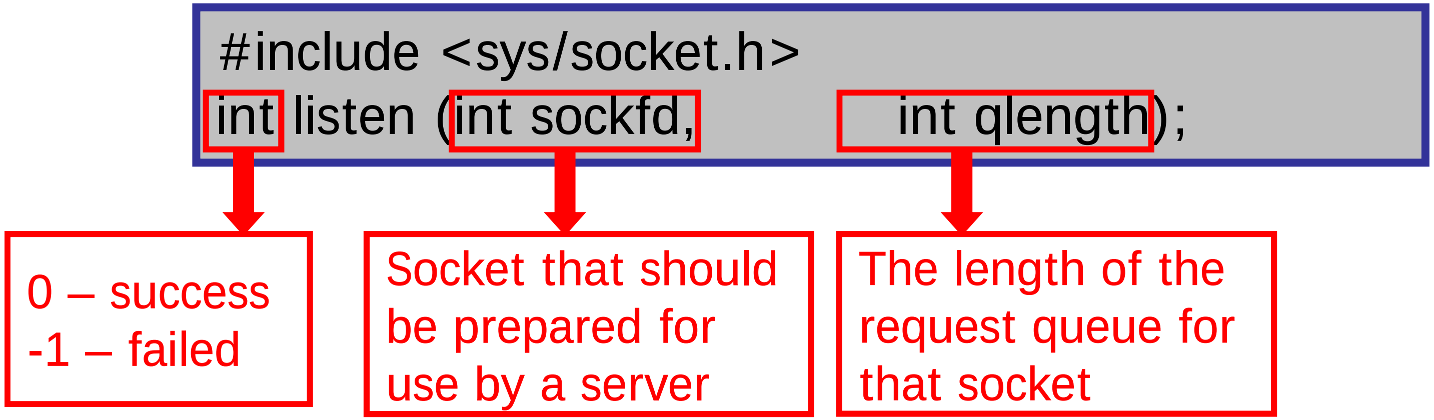
- In Linux, as in most other UNIX systems, programmers can use *read* instead of *recv*, and *write* instead of *send*
  - *int read (sockfd, bptr, buflen)*
  - *int write (sockfd, bptr, buflen)*
- The chief advantage of *send* and *recv* is that they are easier to spot in the code

Other System Calls for sending and receiving data through a socket

- Sending data
  - writev ()
  - sendmsg ()
  - writen()
- Receiving data
  - readv ()
  - recvmsg ()
  - readn()
  - reanline()

System Calls – listen()

- Connection-oriented **servers** call *listen* to place a socket in *passive mode* and make it ready to accept incoming connections
- *Listen* also sets the number of incoming connection requests that the protocol software should enqueue for a given socket while the server handles another request
- It only applies to socket used with **TCP**



```
#include <sys/socket.h>
int listen (int sockfd,         int qlength);
```

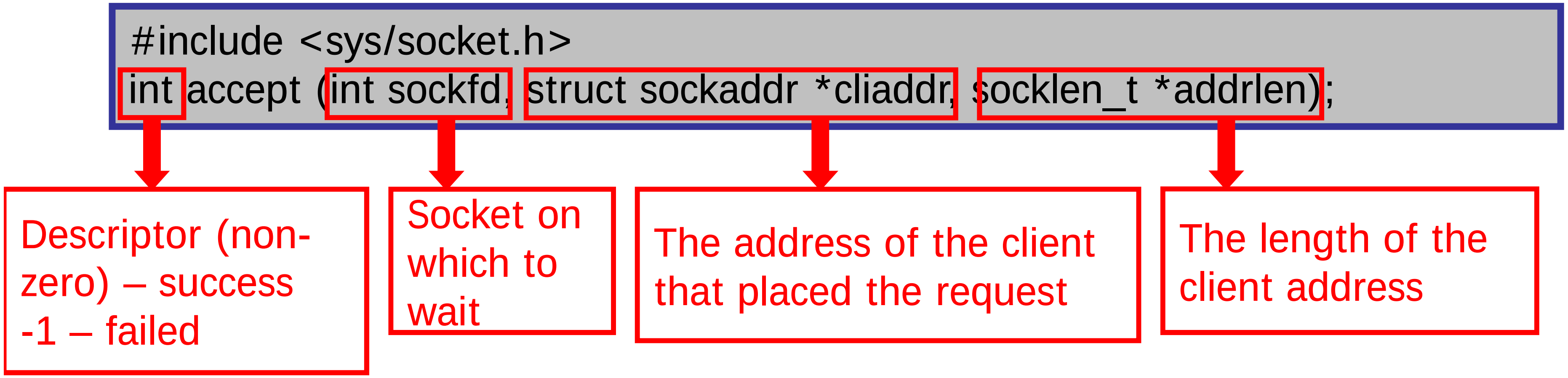
0 – success
-1 – failed

Socket that should be prepared for use by a server

The length of the request queue for that socket

System Calls – accept()

- The **server** calls accept to extract the next incoming request
- *Accept* creates a new socket for each new connection request, and return the descriptor of the new socket to its caller
- *Accept* fills in the structure (*sockaddr*) with the IP address and protocol port number of the remote machine



```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Descriptor (non-zero) – success
-1 – failed

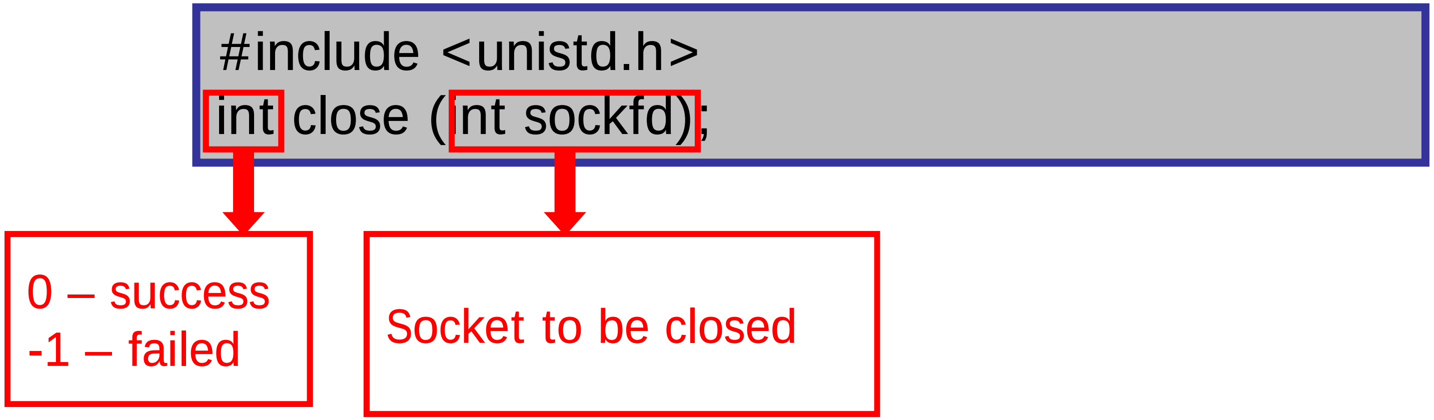
Socket on which to wait

The address of the client that placed the request

The length of the client address

System Calls – close()

- Once a client or server finishes using a socket, it calls *close* to deallocate it
- If several processes share a socket, *close* decrements a reference count and deallocates the socket when the reference count reaches zero
- Any unread data waiting at the socket will be discarded



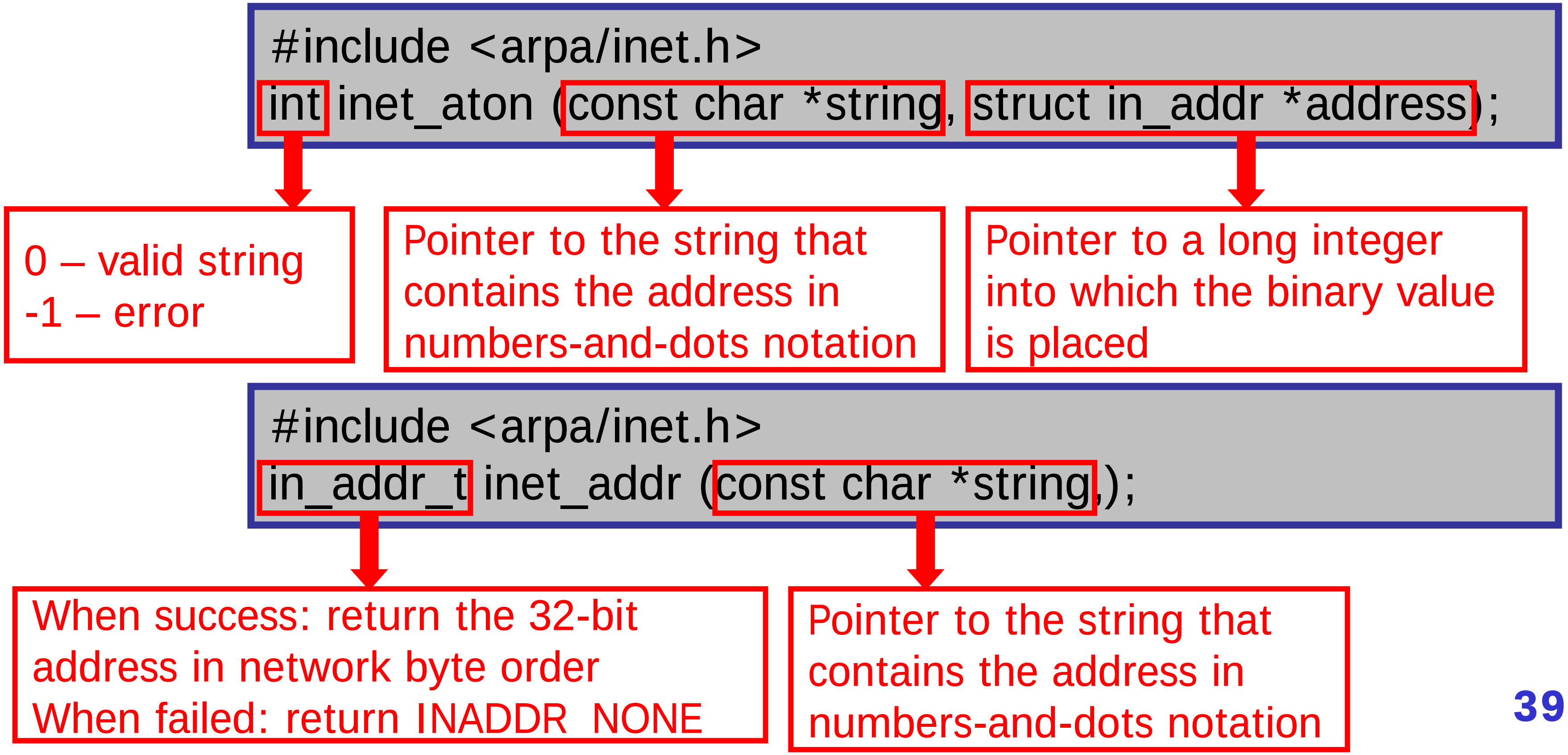
```
#include <unistd.h>
int close (int sockfd);
```

0 – success
-1 – failed

Socket to be closed

System Calls – inet_aton() & inet_addr()

- converts an IP address in numbers-and-dots notation into unsigned long in network byte order



```
#include <arpa/inet.h>
int inet_aton (const char *string, struct in_addr *address);
```

0 – valid string
-1 – error

Pointer to the string that contains the address in numbers-and-dots notation

Pointer to a long integer into which the binary value is placed

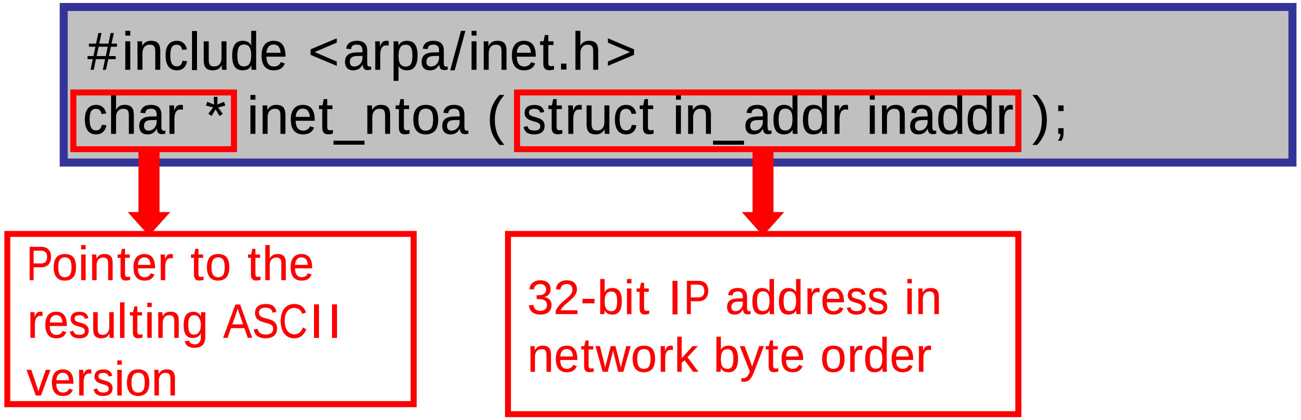
```
#include <arpa/inet.h>
in_addr_t inet_addr (const char *string,);
```

When success: return the 32-bit address in network byte order
When failed: return INADDR_NONE

Pointer to the string that contains the address in numbers-and-dots notation

System Calls – inet_ntoa()

- Mapping a 32-bit integer (an IP address in network byte order) to an ASCII string in dotted decimal format



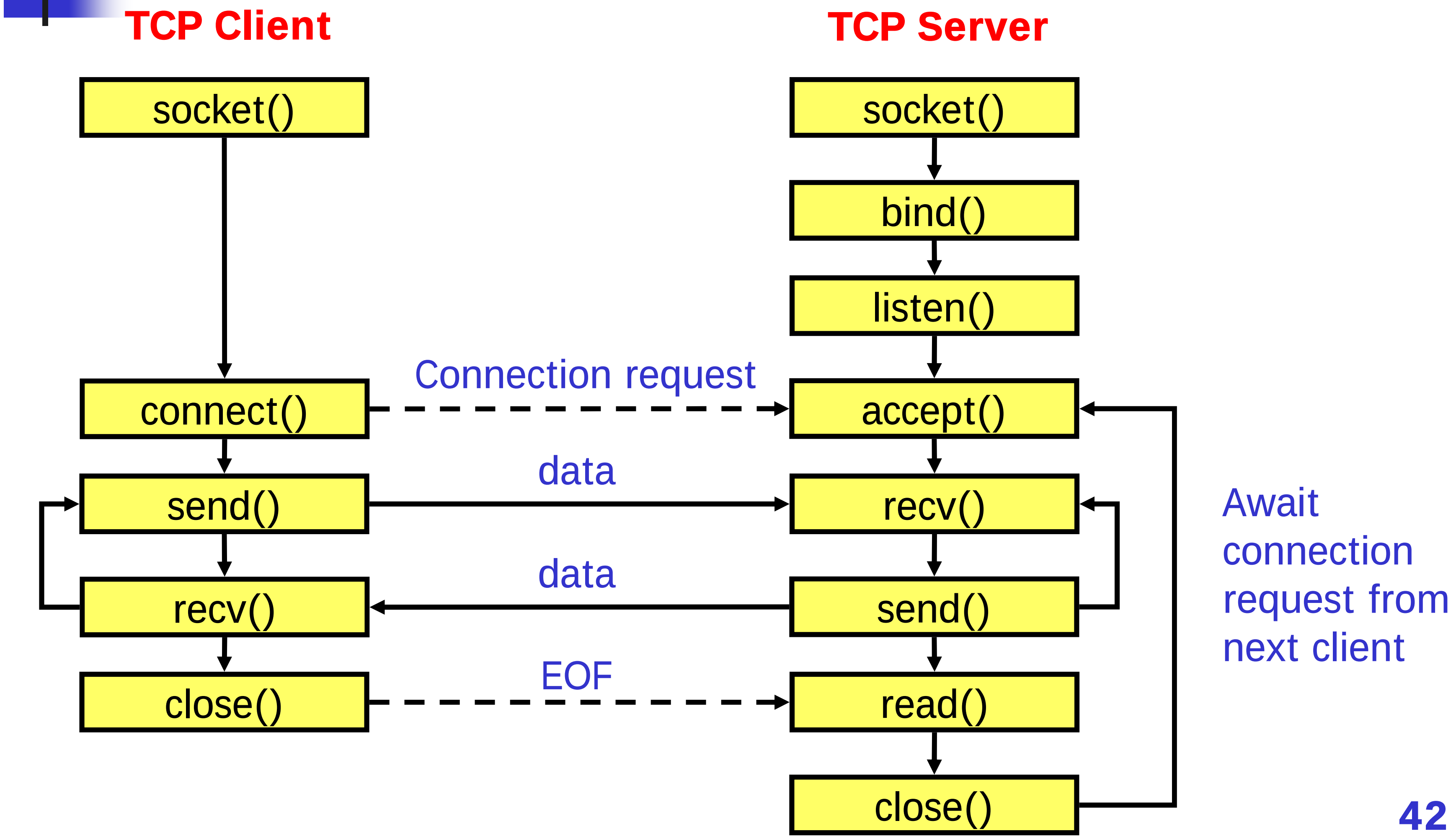
```
#include <arpa/inet.h>
char * inet_ntoa ( struct in_addr inaddr );
```

Pointer to the resulting ASCII version

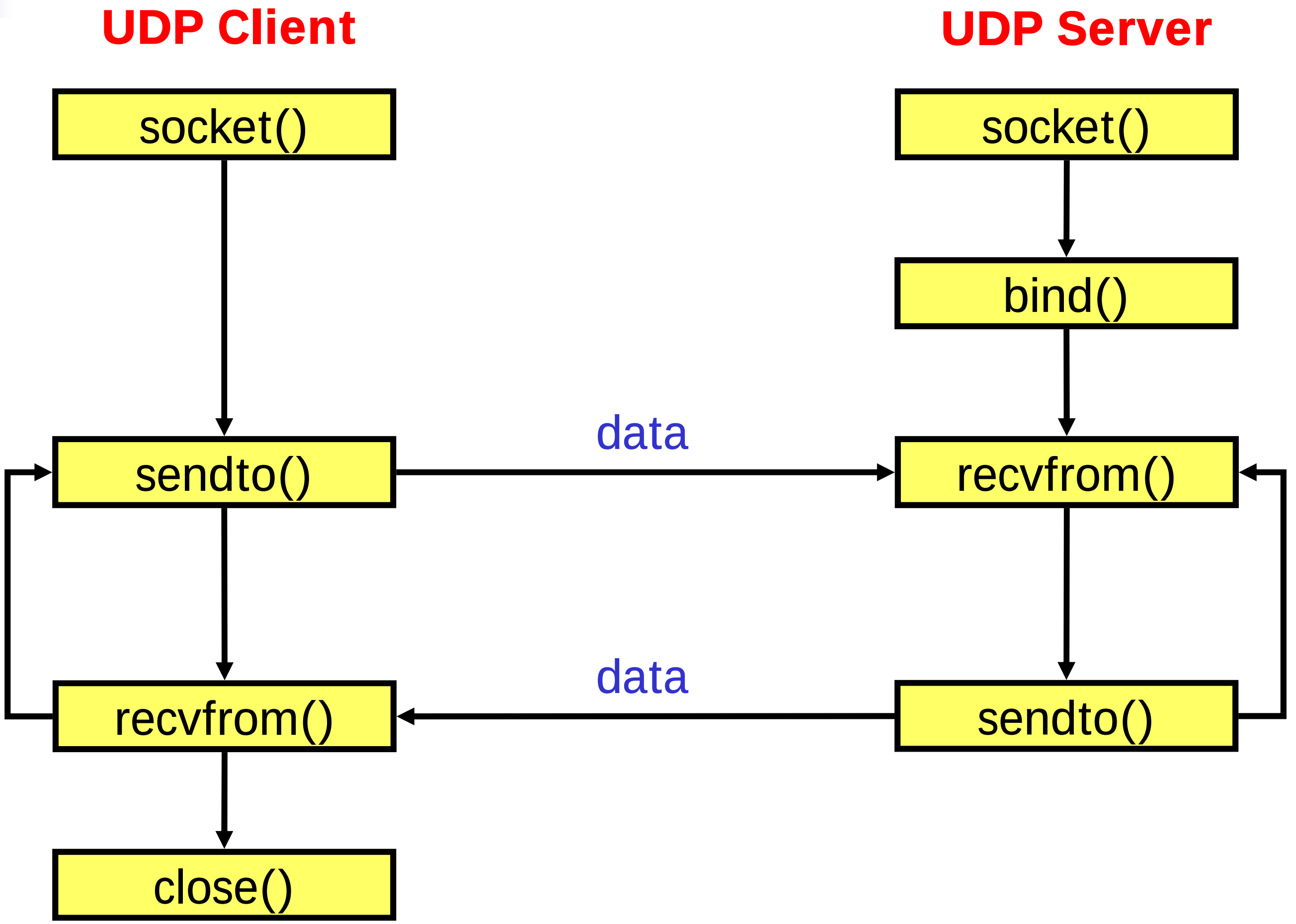
32-bit IP address in network byte order

Introduction to Sockets
Part IV: sample programs

Overview of TCP-based sockets API



Overview of UDP-based sockets API

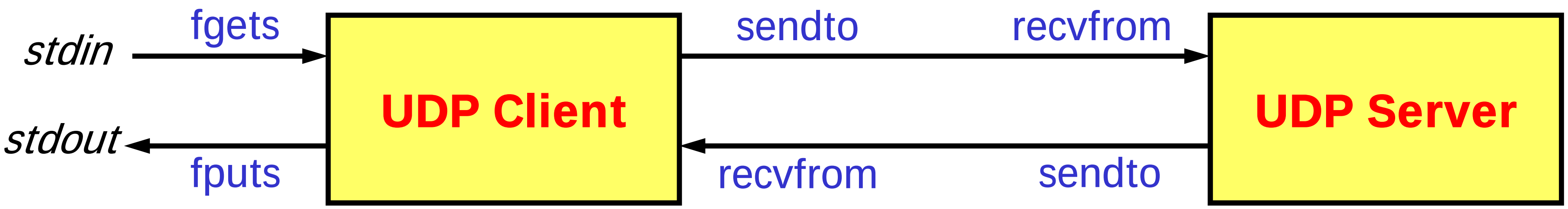


Sample programs

- UDP-based echo service
  - An echo service simply sends back to the originating source any data it receives
  - A very useful debugging and measurement tool
  - UDP Based Echo Service：be defined as a datagram based application on UDP. A server listens for UDP datagrams on UDP port 7. When a datagram is received, the data from it is sent back in an answering datagram.

- Sample programs
  - udpechoclt.c
  - udpechosvr.c

Basic flow of UDP-based echo service



Head part of UDP EchoClient

```
#include <stdio.h> /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), sendto() and
                           recvfrom() */
#include <arpa/inet.h> /* for sockaddr_in and inet_addr() */
#include <stdlib.h> /* for atoi() and exit() */
#include <string.h> /* for memset() */
#include <unistd.h> /* for close() */
```

Initial part of UDP EchoClient

```
#define ECHOMAX 255 /* Longest string to echo */

int main(int argc, char *argv[])
{
   int sock; /* Socket descriptor */
   struct sockaddr_in echoServAddr; /* Echo server address */
   struct sockaddr_in fromAddr; /* Source address of echo */
   unsigned short echoServPort; /* Echo server port */
   unsigned int fromSize; /* In-out of address size
                             for recvfrom() */
   char *servIP; /* IP address of server */
   char *echoString; /* String to send to echo server */
   char echoBuffer[ECHOMAX+1]; /* Buffer for receiving
                                   echoed string */
   int echoStringLen; /* Length of string to echo */
   int respStringLen; /* Length of received response */
```

Argument check part of UDP EchoClient

```
if ((argc < 3) || (argc > 4)) /* Test for correct number of
 arguments */
{
    printf("Usage: %s <Server IP> <Echo Word> [<Echo Port>]\n",
               argv[0]);
    exit(1);
 }

servIP = argv[1]; /* First arg: server IP address (dotted quad) */
echoString = argv[2]; /* Second arg: string to echo */
if ((echoStringLen = strlen(echoString)) > ECHOMAX) /* Check input
                                                        length */

    printf("Echo word too long.\
if (argc == 4)
    echoServPort = atoi(argv[3]); /* Use given port, if any */
else
    echoServPort = 7; /* 7 is the well-known port for echo service */
```

ASCII to integer

I/O part of UDP EchoClient

```
/* Create a datagram/UDP socket */
if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    printf("socket() failed.\n");
/* Construct the server address structure */
memset(&echoServAddr, 0, sizeof(echoServAddr));/*Zero out structure*/
echoServAddr.sin_family = AF_INET; /* Internet addr family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP);/*Server IP address*/
echoServAddr.sin_port = htons(echoServPort); /* Server port */
/* Send the string to the server */
if ((sendto(sock, echoString, echoStringLen, 0,
        (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)))
        != echoStringLen)
    printf("sendto() sent a different number of bytes than expected.\n");
/* Recv a response */
fromSize = sizeof(fromAddr);
if ((respStringLen = recvfrom(sock, echoBuffer, ECHOMAX, 0,
        (struct sockaddr *) &fromAddr, &fromSize)) != echoStringLen)
    printf("recvfrom() failed\n");
```

Generic socket address

Last part of UDP EchoClient

```
if (echoServAddr.sin_addr.s_addr != fromAddr.sin_addr.s_addr)
{
     printf("Error: received a packet from unknown source.\n");
     exit(1);
 }

/* null-terminate the received data */
echoBuffer[respStringLen] = '\0';
printf("Received: %s\n", echoBuffer); /*Print the echoed message*/
close(sock);
exit(0);
}
```

Head part of UDP EchoServer

```
#include <stdio.h> /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), sendto()
                              and recvfrom() */
#include <arpa/inet.h> /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h> /* for atoi() and exit() */
#include <string.h> /* for memset() */
#include <unistd.h> /* for close() */
```

Initial part of UDP EchoServer

```
#define ECHOMAX 255 /* Longest string to echo */

int main(int argc, char *argv[])
{
    int sock; /* Socket */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned int cliAddrLen; /* Length of client address */
    char echoBuffer[ECHOMAX]; /* Buffer for echo string */
    unsigned short echoServPort; /* Server port */
    int recvMsgSize; /* Size of received message */
```

Argument check part of UDP EchoServer

```
if (argc != 2)
{
    printf("Usage: %s <UDP SERVER PORT>\n", argv[0]);
    exit(1);
 }
```

Socket part of UDP EchoServer

```
echoServPort = atoi(argv[1]); /* First arg: local port */

/* Create socket for sending/receiving datagrams */
if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
    printf("socket() failed.\n");
/* Construct local address structure */
memset(&echoServAddr, 0, sizeof(echoServAddr));
echoServAddr.sin_family = AF_INET;
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);
echoServAddr.sin_port =htons(echoServPort);
/* Bind to the local address */
if ((bind(sock, (struct sockaddr *) &echoServAddr,
      sizeof(echoServAddr))) < 0)
    printf("bind() failed.\n");
```

Main loop of UDP EchoServer

```
for (;;) /* Run forever */
{
    /* Set the size of the in-out parameter */
    cliAddrLen = sizeof(echoClntAddr);
    /* Block until receive message from a client */
    if ((recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX,
         0,(struct sockaddr *) &echoClntAddr, &cliAddrLen)) < 0)
        printf("recvfrom() failed.\n");
    printf("Handling client %s\n",
             inet_ntoa(echoClntAddr.sin_addr));
    /* Send received datagram back to the client */
    if ((sendto(sock, echoBuffer, recvMsgSize, 0,
         (struct sockaddr *) &echoClntAddr,
          sizeof(echoClntAddr))) != recvMsgSize)
        printf("sendto() sent a different number of bytes
                than expected.\n");
 }
```

Run the Sample Programs (1)

- Give correct arguments

Server process window

```
[shiyan@localhost 20071022]$ ./udpechosvr
Usage: ./udpechosvr <UDP SERVER PORT>
```

Client process window

```
[shiyan@localhost 20071022]$ ./udpechoclt
Usage: ./udpechoclt <Server IP> <Echo Word> [<Echo Port>]
```

Run the Sample Programs (2)

- Use correct username

Server process window

```
[shiyan@localhost 20071022]$ ./udpechosvr 7
bind() failed.
```

Note: binding the port number less than 1024 requires root authority

Client process window

```
[shiyan@localhost 20071022]$ ./udpechoclt 192.168.1.253 hello
```

Run the Sample Programs (3)

- Successful running using root

Server process window

```
[root@localhost 20071022]# ./udpechosvr 7
Handling client 192.168.1.253
```

Client process window

```
[root@localhost 20071022]# ./udpechoclt 192.168.1.253 hello
Received: hello
[root@localhost 20071022]#
```

Run the Sample Programs (4)

- Successful running using other username

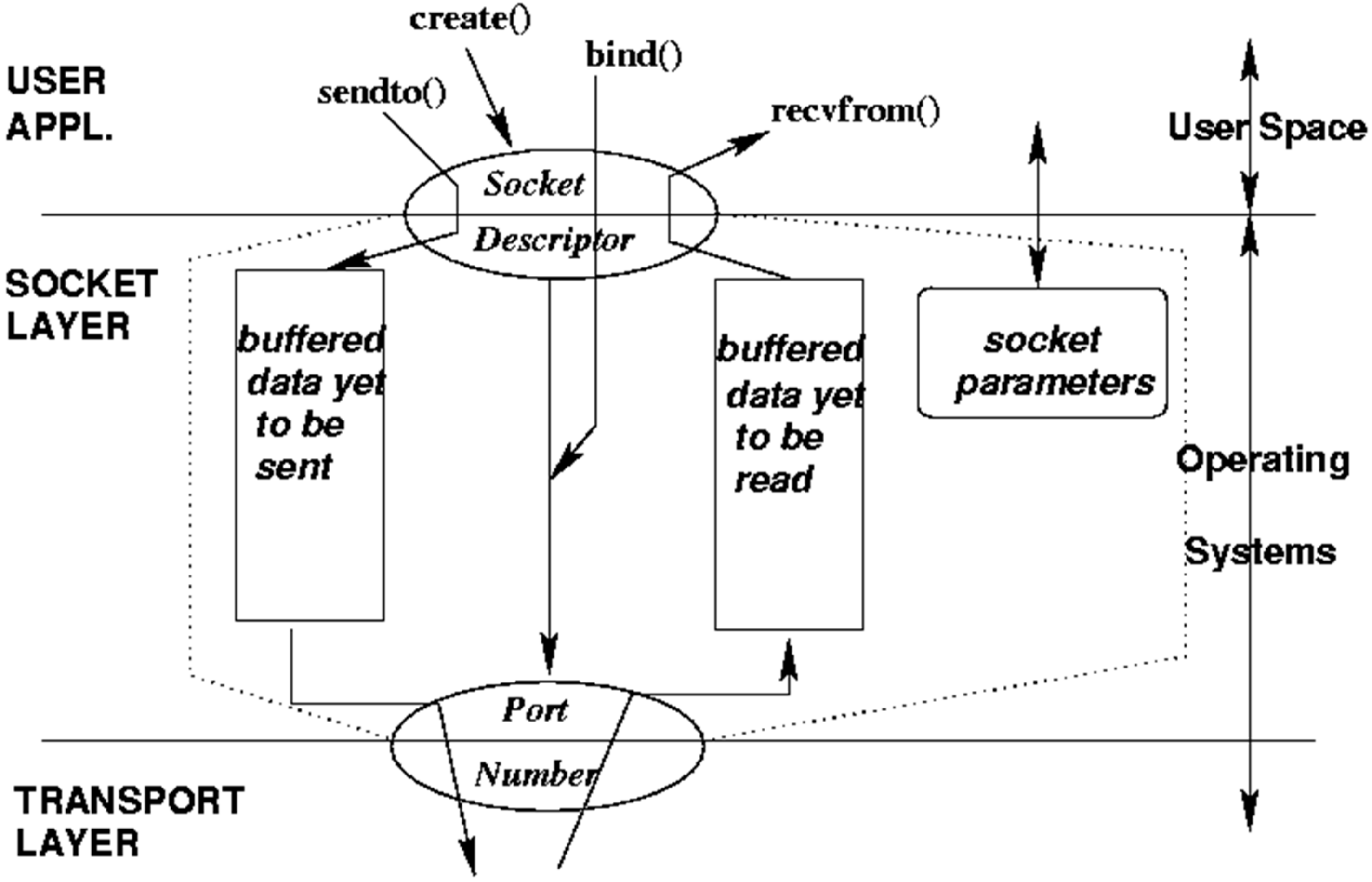
Server process window

```
[shiyan@localhost 20071022]$ ./udpechosvr 1500
Handling client 192.168.1.253
```

Client process window

```
[shiyan@localhost 20071022]$ ./udpechoclt 192.168.1.253 hello 1500
Received: hello
[shiyan@localhost 20071022]$
```

Summary:
Conceptual View of Socket


create()
bind()
USER
APPL.
sendto()
recvfrom()
User Space
Socket
Descriptor
SOCKET
LAYER
buffered
data yet
to be
sent
buffered
data yet
to be
read
socket
parameters
Operating
Systems
Port
Number
TRANSPORT
LAYER