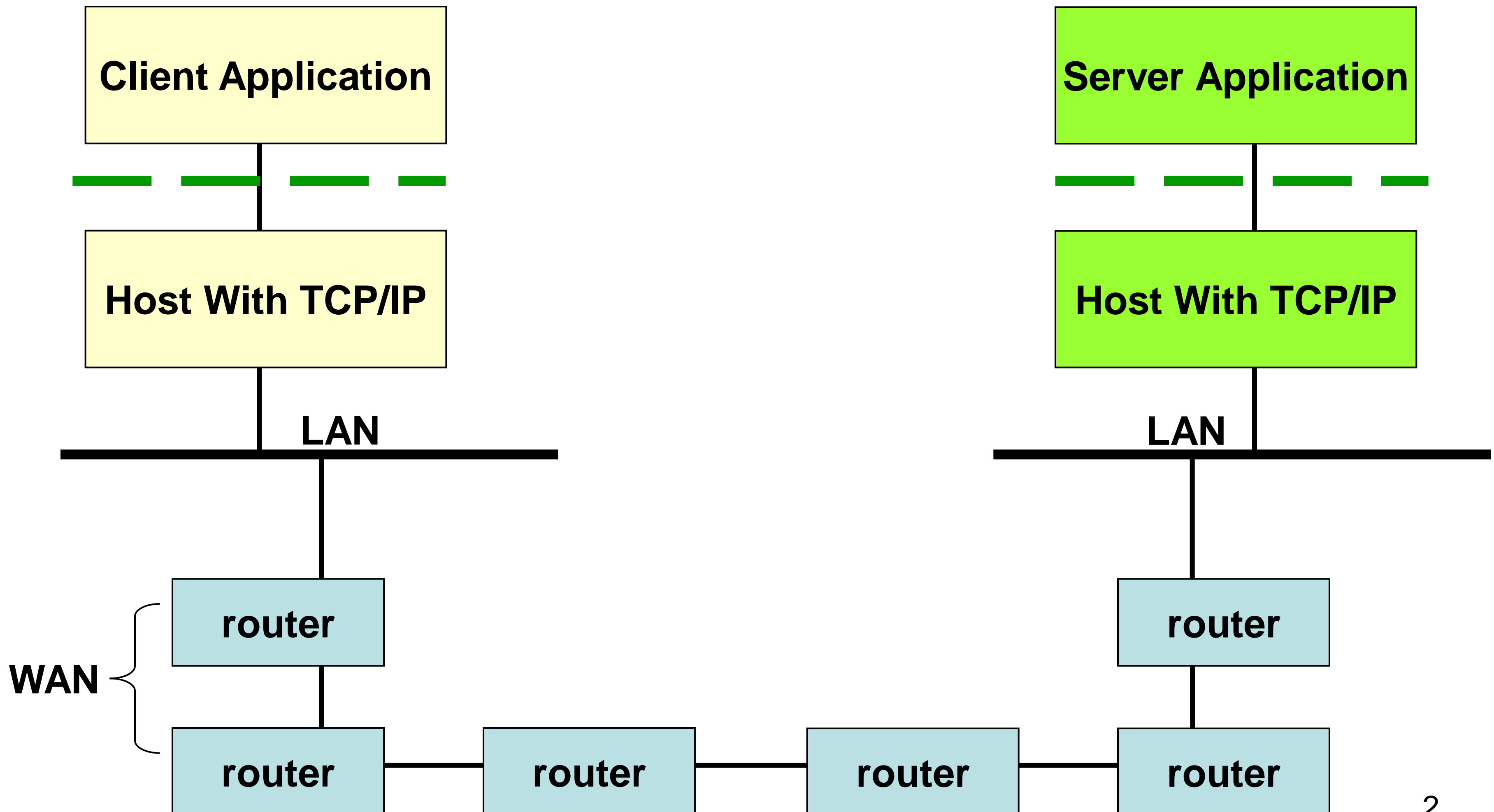
Socket Programming

Dr. Yeali S. Sun
National Taiwan University

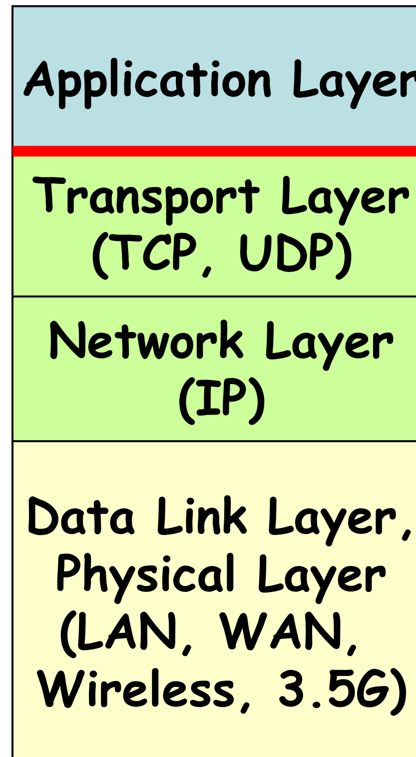
Client-Server Communication Paradigm

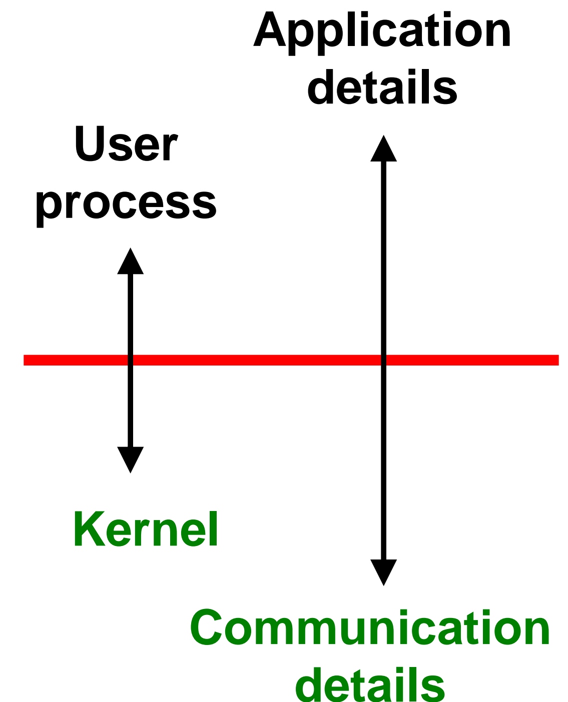


Application Interface to Protocols

Network Programming

- Network applications use TCP/IP to communicate with each other.
- TCP/IP protocol software resides in the computer's operating system.
- An application program interacts with the operating system to request service.



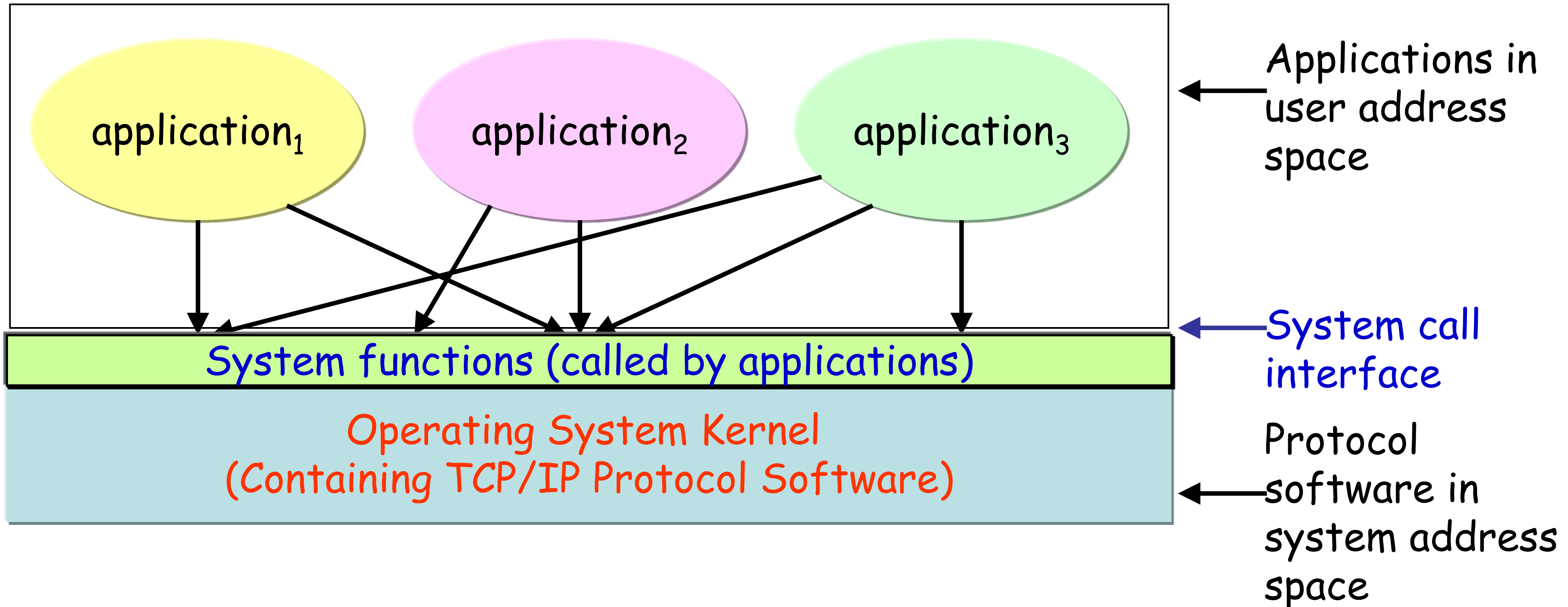


TCP/IP Network Programming APIs

- In practice, a few APIs exist.
- Berkeley UNIX Sockets
  - Initiated by ARPA (Advanced Research Project Agency) in early 1980s
  - Done by the University of California, Berkeley
  - Included in release 4.1 of the Berkeley Software Distribution (**bsd**)
  - Has been adopted by many systems, including Linux (a *de facto* standard)
  - Known as *socket API* (or socket interface, sockets.)
- Microsoft Windows Sockets
  - A variant of socket API
- AT&T TLI (Transport Layer Interface)
  - for System V UNIX

The Socket Interface

System Calls



- System calls are mechanisms that OS uses to transfer control between application and the operating system procedures.
- To a programmer, system calls look and act like function calls.
- When received a system call, OS directs the call to an internal procedure that performs the requested operation.

The basic I/O operations in Linux

- An application program calls open to initiate input or output.
- The system returns an integer called a file descriptor that the application uses in further I/O operations
- Three arguments
  - the *name* of a file or device to open
  - A set of bit *flags* that controls special cases (e.g., create one if not exists)
  - An *access* mode (e.g., read, write, etc.)

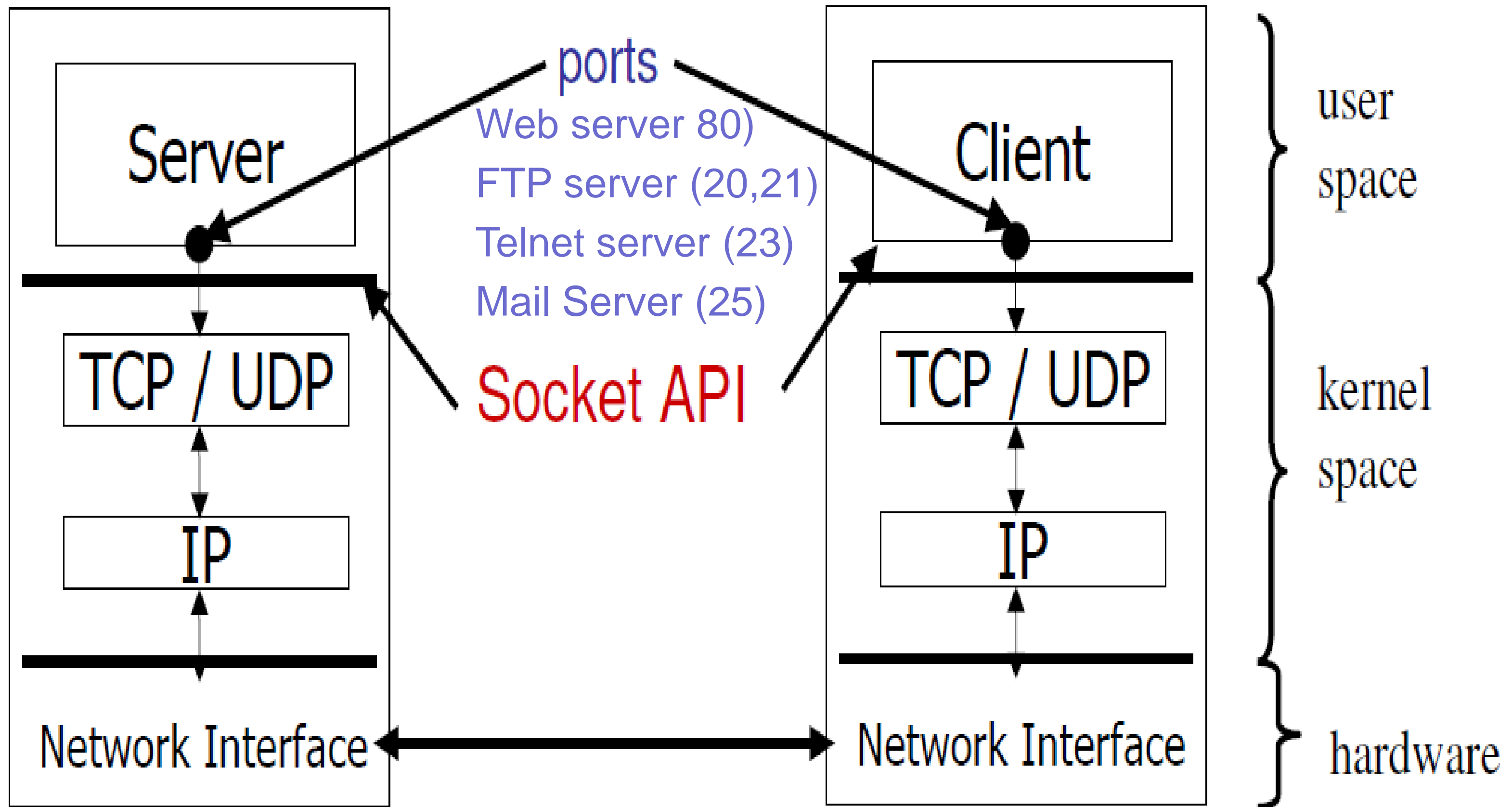
- Example

```
int desc;
desc =
   open("filename",
   O_RDWR, 0);

read (desc, buffer,
   128); //bytes

close (desc);
```

Socket API (1/2)

- Socket API is a set of functions and the parameters that each function requires and the *semantics* of the operation it performs.
- Follow conventional I/O primitives notation and semantics
  - Use basic I/O functions whenever possible
  - Add additional functions for those operations that cannot be expressed conveniently
- Allow multiple families of protocols
  - E.g., TCP/IP protocol family – PF_INET

Socket API (2/2)



Socket API: Functionality

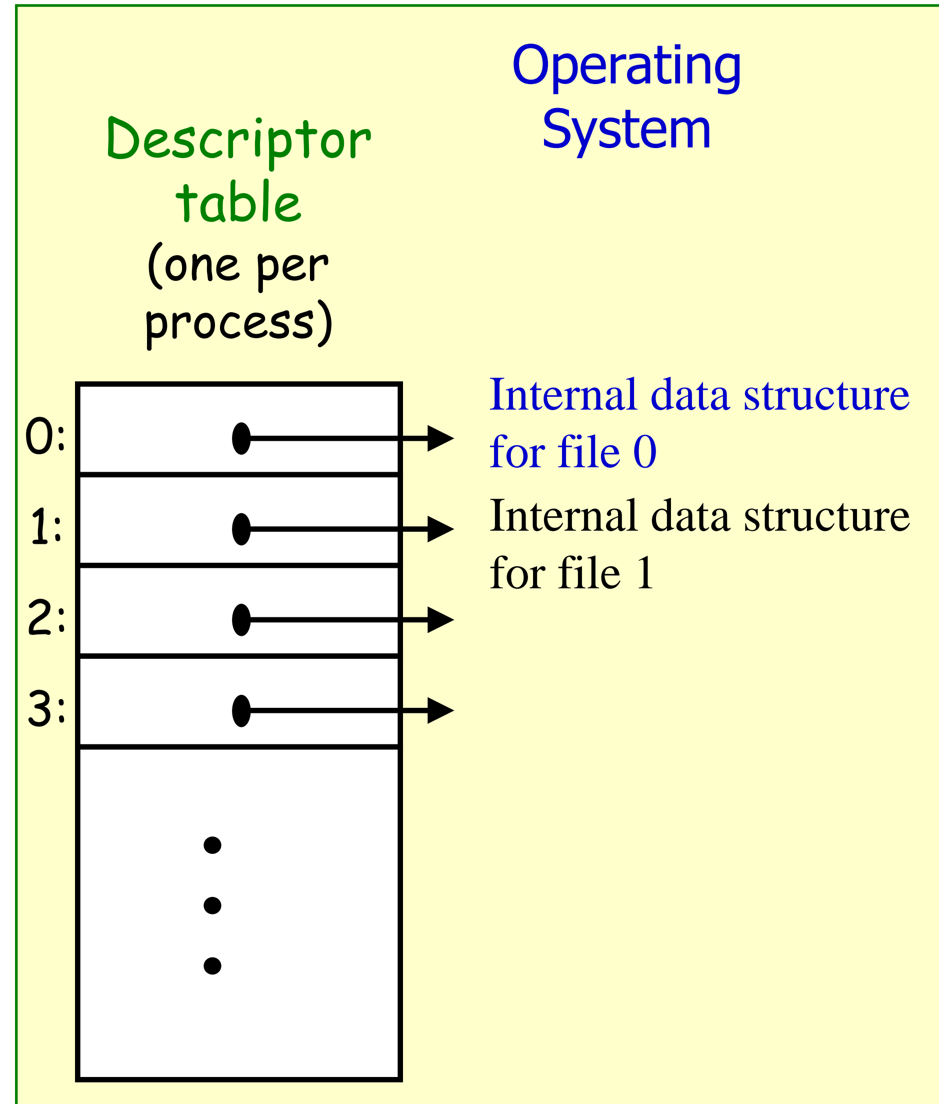
- Allocate local resources for communication
- Specify local and remote communication endpoints
- Initiate a connect (client side)
- Send a datagram (client side)
- Wait for an incoming connection (server side)
- Send or receive data
- Determine when data arrives
- Generate urgent data
- Handle incoming urgent data
- Terminate a connection gracefully
- Handle connection termination from the remote site
- Abort communication
- Handle error conditions or a connection abort
- Release local resource when communication finishes.

The basic I/O operations in Linux

open	Prepare a device or a file for input and output operations
close	Terminate use of a previously opened device or file
read	Obtain data from an input device or file, and place it in the application program's memory
write	Transmit data from the application program's memory to an output device or file
lseek	Move to a specific position in a file or device (e.g., disk)
ioctl	Control a device or the software used to access it (e.g., specify buffer size or change character set mapping)

Sockets for Network Communication

- OS implements **file descriptors** as *an array of pointers to internal data structures*.
- OS maintains a *separate* file descriptor table for each *process*.
- Socket descriptor
  - When a process opens a socket, OS places a pointer to the internal data structure for that socket.
  - It is in the same process' descriptor table as file descriptors.
  - OS returns the table index (i.e. the socket descriptor) to the calling program.



Summary of Socket Calls (1/2)

socket	Create a descriptor for use in network communication
connect	Connect to a remote peer (client)
send (write)	Send outgoing data across a TCP connection or a UDP datagram
recv (read)	Acquire incoming data from a TCP connection or the next incoming UDP datagram
close	Terminate comm. and de-allocate a descriptor
bind	Bind a local IP addr. and protocol port to a socket
listen	Place the socket **in passive mode** and set **the # of incoming TCP conn**. the system will enqueue (server)
accept	Accept the next incoming conn. (server)

Summary of Socket Calls (2/2)

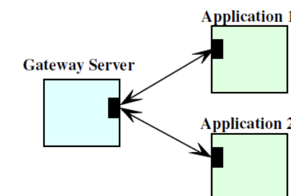
recvmsg	Receive next incoming UDP datagram (variation of recv).
recvfrom	Receive next incoming UDP datagram and record its source endpoint address
sendmsg	Send an outgoing UDP datagram
sendto	Send an outgoing UDP datagram, usually to a prerecorded endpoint address
shutdown	Terminate a TCP conn. in one or both directions
getpeername	After a conn. arrives, obtain the remote machine's endpoint address from a socket
getsockopt	Obtain the current options for a socket
setsockopt	Change the options for a socket

Two essential types of sockets

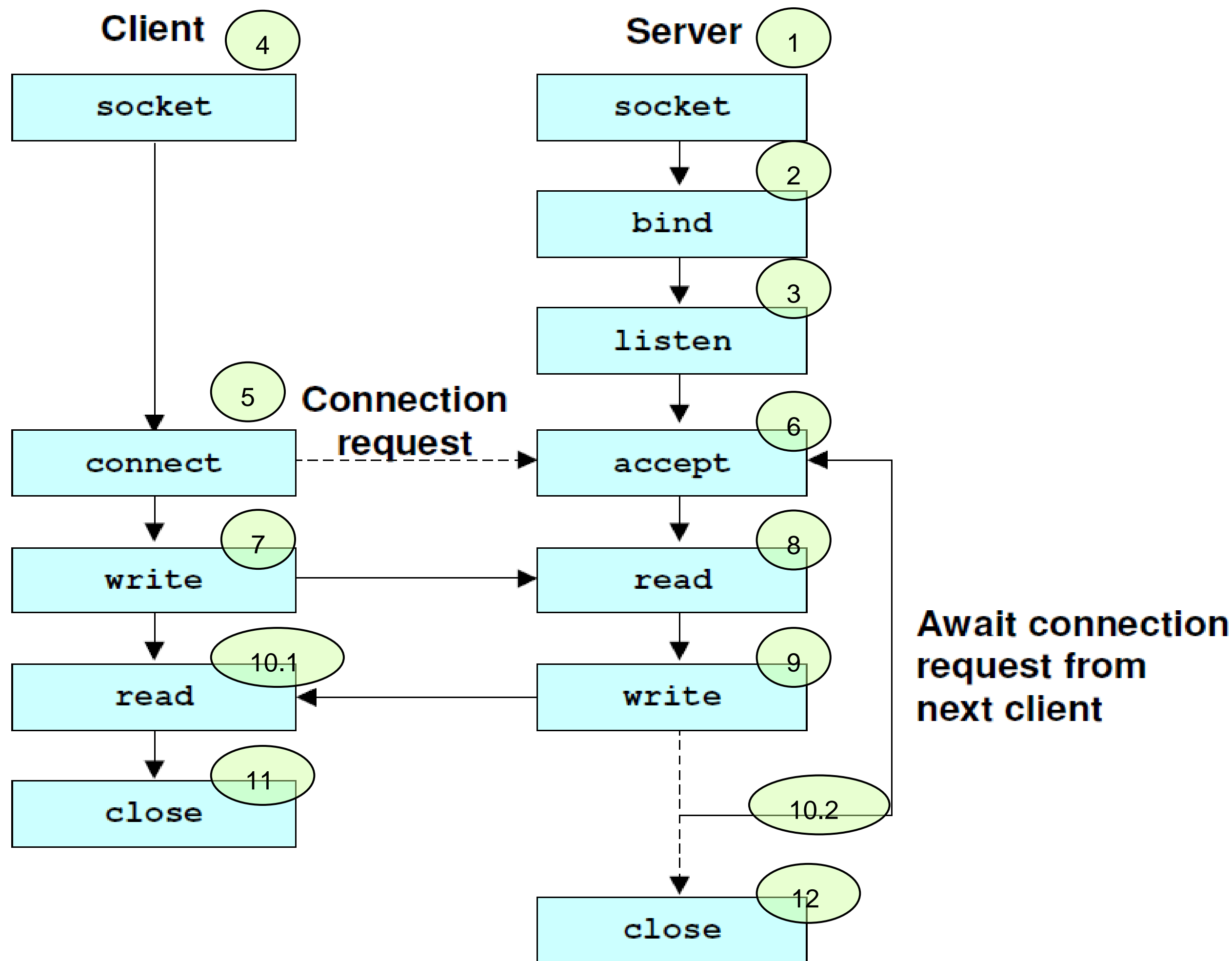
- SOCK_STREAM
  - **TCP**
  - connection-oriented
  - Reliable delivery
  - in-order guaranteed
  - bidirectional

- SOCK_DGRAM
  - **UDP**
  - no notion of "connection"-

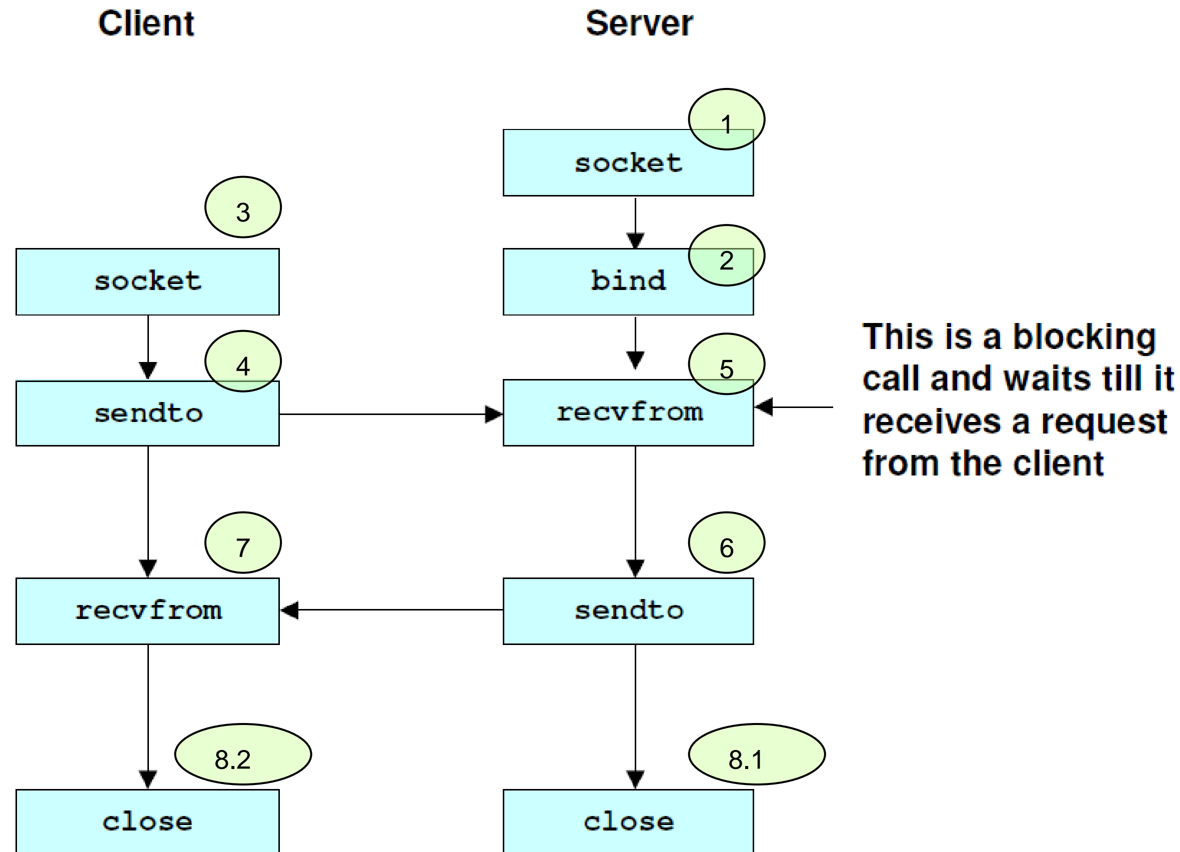
  app indicated dest. for each packet
  - unreliable delivery
  - no order guarantees
  - can send or receive



Sequence of calls made by a "client" and a "server" using TCP

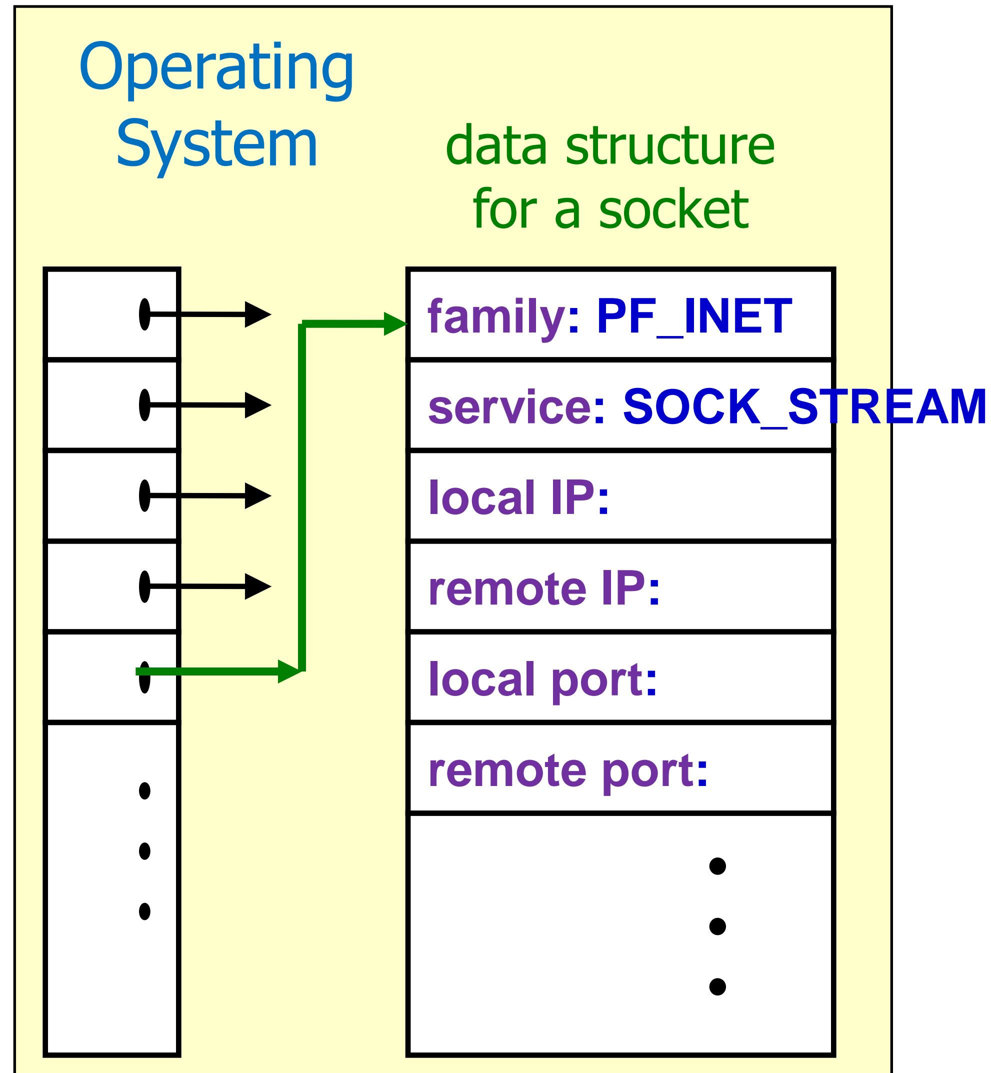


Sequence of calls made by a "client" and a "server" using UDP



System data structures for socket()

- socket(): create a new socket
- A new data structure is created by OS to hold the info. for communication
- A *new* descriptor table entry is created to contain a pointer to the data structure.



Making a socket active or passive

- Server
  - Configure a socket to *wait* for an incoming connection
  - The socket is said to be *passive*
- Client
  - Configure a socket to *initiate* a connection to server
  - The socket is said to be *active*

Predefined symbolic constants and data structures for socket calls

- SOCK_DGRAM, SOCK_STREAM
- #include <sys/types.h>
- #include <sys/socket.h>

Specifying an endpoint address

- Address family
  - e.g., AF_INET (PF_INET)
- Each protocol family defines its own representation of its endpoint address

TCP/IP endpoint address

```
struct sockaddr_in {                          /* struct to hold an address    */
    u_char              sin_len;              /* total length                 */
    u_short             sin_family;           /* type of address: 2-byte      */
    u_short             sin_port;             /* protocol port number         */
    struct in_addr      sin_addr;             /* IP address (declared to be */
                                              /*  u_long on some systems) */
    char        sin_zero[8];                  /* unused (set to zero)         */
};
```

Algorithms for Client Software Design

Introduction

Client program

- How to initiate communication?
- Select TCP/IP protocol and address families
- How to use TCP or UDP as the transport layer protocol?
- How to contact a server?
- How to use socket calls to interact with the protocols?

Algorithm: A "TCP Client" to form a connection to a server for communication

1. Find the IP address and protocol port number of the server with which communication is desired
2. Allocate a socket
3. Specify that the connection needs an arbitrary, unused protocol port on the local machine, and allow TCP to choose one
4. Connect the socket to the server
5. Communication with the server using the application-level protocol (this usually involves sending requests and awaiting replies)
6. Close the connection

Step 1: Identify the location of a server

- Have server's domain name or IP address as a constant when the program is compiled
  - inflexible
- Find the server when the program is invoked
  - as an input argument
  - from stable storage (e.g., a file), or
  - using a separate protocol to find a server (e.g., multicast, broadcast, etc)

➢ Make the client program more general

➢ Make it possible to change server locations

Step 1-A: Look up a domain name

- socketaddr_in requires a 32-bit IP address in binary
- Socket APIs that convent a dotted decimal address (1.2.3.4) into a 32-bit IP address in binary
  - inet_addr
    - Take an ASCII string address and return the equivalent IP address in binary
  - gethostbyname
    - Take an ASCII string address and return the address of a hostent structure

```
struct sockaddr_in {
      u_char                 sin_len;
      u_short                sin_family;
      u_short                sin_port;
      struct in_addr         sin_addr;
      char                   sin_zero[8];
};
```

gethostbyname()

```
struct hostent {
    char      *h_name;           /* official host name       */
    char      **h_aliases;       /* other aliases            */
    int        h_addrtype;       /* address type             */
    int        h_length;         /* address length           */
    char      **h_addr_list;     /* list of address          */
};
#define h_addr h_addr_list[0]    /* for backward compatibility
```

- *Lists* of host names and aliases (a host may have more than one interface)

Look up a domain name: sample code

```
struct hostent *hptr;
char    *examplename = “merlin.cs.purdue.edu”;

If ( hptr = gethostbyname(examplename) ) {
        /* IP address is now in hptr->h_addr */
} else {
        /* error in name – handle it */
}
```

```
struct hostent {
    char      *h_name;          /* official host name     */
    char     **h_aliases;       /* other aliases          */
    int        h_addrtype;      /* address type           */
    int        h_length;        /* address length         */
    char     **h_addr_list;     /* list of address        */

}; // #define h_addr h_addr_list[0]     /* for backward compatibility
```

Step 1-B: Look up a well-known port by name

- Look up the protocol port for a service
- get**serv**byname(string service, string protocol)

```
struct servent {
    char      *s_name;       /* official service name   */
    char     **s_aliases;    /* other aliases           */
    int        s_port;       /* port for this service   */
    char      *s_proto;      /* protocol to use         */
};
```

Step 1-B: Look up a well-known port by name: sample code

```
struct servent *sptr;
char    *examplename = “merlin.cs.purdue.edu”;

If ( sptr = getservbyname(“smtp”, “tcp”) ) {
        /* port number is now in sptr->s_port */
} else {
        /* error in name – handle it */
}
```

```
struct servent {
    char      *s_name;          /* official service name   */
    char     **s_aliases;       /* other aliases           */
    int        s_port;          /* port for this service   */
    char      *s_proto;         /* protocol to use         */
};
```

Port number and network byte order

- getservbyname() returns the protocol port in network byte order
  - It is in the form for use in sockaddr_in
- Network byte order vs. byte order in local machine(!)

Step 1-C: Look up a protocol by name

- A protocol name is mapped to an integer constant (e.g., TCP:6, UDP:17)
- getprotobyname()

```
struct protoent {
   char        *p_name;           /* official protocol name   */
   char      **p_aliases;         /* list of aliases allowed   */
   int         *p_proto;          /* official protocol number*/
};
```

Step 1-C: Look up a protocol by name: sample code

```
struct protoent *pptr;


If ( pptr = getprotobyname(“udp”) ) {
        /* official protocol number is now in pptr->p_proto */
} else {
        /* error in name – handle it */
}
```

Step 2: Allocate a Socket

```
#include <sys/types.h>
#include <sys/socket.h>

int sd;          /* socket descriptor*/
sd = socket(PF_INET, SOCK_STREAM, 0);
```

Step 3: Choose a local protocol port

- The socket call allows an application to leave the local IP address *unfilled*
- TCP/IP software will choose a local one *automatically* at the time the client connects to a server.

Step 4: Connect the socket to the server

- connect()
  - Allow a client to initiate a connection
  - Return value: 0: success; 1: failure

```
retcode = connect(sd, remaddr, remaddrlen);
```

- sd: socket descriptor (socket())
- remaddr: remote endpoint of the connection of type sockaddr_in
- remaddrlen: in bytes

connect()

Performs four tasks

- Test to ensure the specified socket is valid and has not been connected.
- Fills in the remote endpoint address in the socket
- Choose a local endpoint address for the connection if not having one
- Initiate a connection and return a value whether succeeded or not

Step 5: Communicating with server using TCP: sample code

```
#define BLEN 120                      /* buffer length to use */
char    *req = "request for some port";
char    buf[BLEN];                    /* buffer for answer */
char    *bptr;                        /* pointer to buffer */
int     n;                            /* number of bytes read */
int     buflen;                       /* space left in buffer */

bptr = buf;
buflen = BLEN;

send(sd, req, strlen(req), 0 ); /* Send request */

/* read response (may come in many pieces) */

while ( ( n = recv(sd, bptr, buflen, 0) ) > 0 ) {
        bptr += n;
        buflen -= n;
}
```

Receiving a response from a TCP connection

- TCP is stream-oriented
  - deliver the sequence of bytes that the sender transmits;
  - Do not guarantee to deliver to receiver in the same grouping as they were sent.
- TCP may choose to accumulate many bytes in its output buffer before sending a segment.

Step 6: Closing a TCP connection

- close()
  - Terminate the connection gracefully and deallocate the socket.
- TCP is a two-way communication
  - Terminate a connection requires coordination among the client and the server
- Partial close - shut down a TCP connection in one direction

  errcode = shutdown(sd, direction)

  - direction: an integer (0: no further input is allowed; 1: no further output is allowd; 2: shutdown in both directions)
  - Client finishes sending may use shutdown()
  - The server receives an end-of-file signal
  - After sending the last response, it can close the connection.

Algorithm: A "UDP Client" to form a connection to a Server for communication

1. Find the IP address and protocol port number of the server with which communication is desired
2. Allocate a socket
3. Choose an arbitrary, unused protocol port on the local machine, or allow client to choose one
4. Specify the server to which message must be sent
5. Communication with the server using the application-level protocol (this usually involves sending requests and awaiting replies)
6. Close the connection

Connected and unconnected UDP sockets

- A client application can use a UDP socket in one of two basic modes: connected and unconnected.
- Connected mode
  - The client calls connect() to specify a remote endpoint address
  - Client can send and receive messages without specifying the remote address repeatedly.
  - Suitable for client app that interacts with *only one* server at a time
- Unconnected mode
  - Does **not** connect the socket to a specific remote endpoint
  - It specifies the remote destination each time it sends a message
  - Suitable for interacting with *multiple* servers

Communicating with a server with UDP

- connect() with SOCK_DGRAM
  - Do not test validity or reachability of the remote endpoint address
  - It records the remote endpoint info in the socket data structure
- UDP
  - message transfer
  - send(), recv()
- close()
  - Do not inform the remote endpoint
- shutdown()
  - For a connected UDP
  - Stop further transmission in a given direction
  - Again, no control message is sent to the other side.

Discussion

- Client applications using UDP must handle reliability functions themselves if needed.
- Reliability techniques, e.g.,
  - Packet sequencing
  - Acknowledgements
  - Timeouts
  - Retransmission

Example Client Software

A Procedure to Form Connections

```
/* connectsock.c - connectsock */

#include <sys/types.h>
#include <sys/socket.h>

#include <netinet/in.h>
#include <arpa/inet.h>

#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#ifndef        INADDR_NONE
#define INADDR_NONE        0xffffffff
#endif         /* INADDR_NONE */

extern int     errno;

int  errexit (const char *format, ...);
```

```
/*------------------------------------------------------------------------
 * connectsock – allocate & connect a socket using TCP or UDP
 *------------------------------------------------------------------------*/
int  connectsock( const char *host, const char *service, const char
    *transport )
/*  host            - name of host to which connection is desired
    service         - service associated with the desired port
    transport       - name of transport protocol to use ("tcp" or "udp") */
{
    struct hostent *phe;      /* pointer to host information entry */
    struct servent *pse ;     /* pointer to service information entry */
    struct protoent *ppe;     /* pointer to protocol information entry */
    struct sockaddr_in sin;  /* an Internet endpoint address */
    int sd, Type;                          /* socket descriptor and socket type */

    memset(&sin, 0, sizeof(sin));   /* clean up */
    sin.sin_family = AF_INET;        /* address family: Internet */
```

```
/* Map service name to port number */
    if ( pse = getservbyname(service, transport) )
            sin.sin_port = pse -> s_port;
    else if ((sin.sin_port = htons((unsigned short)atoi(service))) == 0)
            errexit(“can’t get \ “%s\” service entry\n”, service);

/* Map host name to IP address, allowing for dotted decimal */
    if ( phe = gethostbyname(host) )
            memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
    else if ((sin.sin_addr.s_addr=inet_addr(host)) == INADDR_NONE)
            errexit(“can’t get \ “%s\” host entry\n”, host);


/* Map transport protocol name to protocol number */
    if ( (ppe = getprotobyname(transport)) == 0)
             errexit(“can’t get \ “%s\” protocol entry\n”, transport);

/* Use protocol to choose a socket type */
    if(strcmp(transport, “udp”) == 0)
            type = SOCK_DGRAM;
    else
            type = SOCK_STREAM;
```

```
/* Allocate a socket */
    sd = socket(PF_INET, type, ppr->p_proto);
    if ( sd < 0 )
            errexit("can't create socket: %s\n", strerror(errno));

/* Connect the socket */
    if (connect(sd, (struct sockaddr *)&sin, sizeof(sin)) < 0)
          errexit( "can't connect to %s.%s: %s\n" , host, service ,
          strerror(errno) );

    return sd;
}
```

Algorithms for TCP Server Software Design

Server architecture

- SERVER SIDE

socket

bind

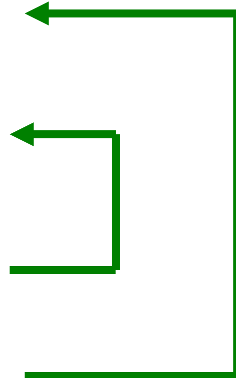
listen

accept

recv

send

close



Create a socket

Bind the socket to a well-known port to receive requests

Enter an infinite loop to accept the next request from clients

Process the request

Formulate a reply

Send reply back

bind()

- sockddr_in – IP address and port number
- getportbyname
  - Used by a server to map a service name into the corresponding well-known port number
- INADDR_ANY (a socket interface constant)
  - To allow a multi-homed hosts and routers to have a single server accept incoming communication addressed to any of the hosts’s IP addresses

listen()

- It has an input argument – specifying the length of an internal **request queue** for the socket
  - Each incoming "TCP connection request"
- Place the socket in **passive** mode

- SERVER SIDE
  - socket
  - bind
  - listen
  - accept
  - recv
  - send
  - close

accept()

- It obtains the next incoming connection request (i.e., extract the request from the request queue)
- It returns a socket descriptor to be used for the new connection.

- Once accepted the connection, use recv() (read()) to obtain application protocol requests from the client
- Use send() (write()) to send replies back.
- Use close() to release the socket

- SERVER SIDE

  socket

  bind

  listen

  accept

  recv

  send

  close

Server architecture: concurrent vs. iterative

- Iterative server
  - Process **one** request at a time
- Concurrent server
  - Handle **multiple** requests at one time, i.e., permitting multiple requests to proceed concurrently
- Multiple threads of execution
  - Each **thread** handles one request

Four types of servers

Low
Request
Processing time

Iterative connectionless	Iterative connection-oriented
Concurrent connectionless	Concurrent connection-oriented

Iterative connectionless

Client

- connect() – to specify a server's address
- write() – to send data (internal data structure contains both two endpoints address)

Server

- recvfrom()
  - Server uses to receive the sender's address

```
retcode = recvfrom(s, buf, len,
  flags, from, fromlen);
```

- Uses an unconnected socket
- sendto() to specify both a datagram to be sent and an address to which it goes.

```
retcode = sendto(s, message,
  len, flags, toaddr, toaddrlen);
```

  - s: unconnected socket
  - Generates reply addresses explicitly use

Four types of servers

Low
Request
Processing time

Iterative connectionless	Iterative connection-oriented
Concurrent connectionless	Concurrent connection-oriented

Concurrent Sever: Goals

- Provide faster response times to multiple clients
- Suitable for applications that
  - form a response required significant I/O
  - *diverse* processing times
  - server executed on *a multi-processor computer*

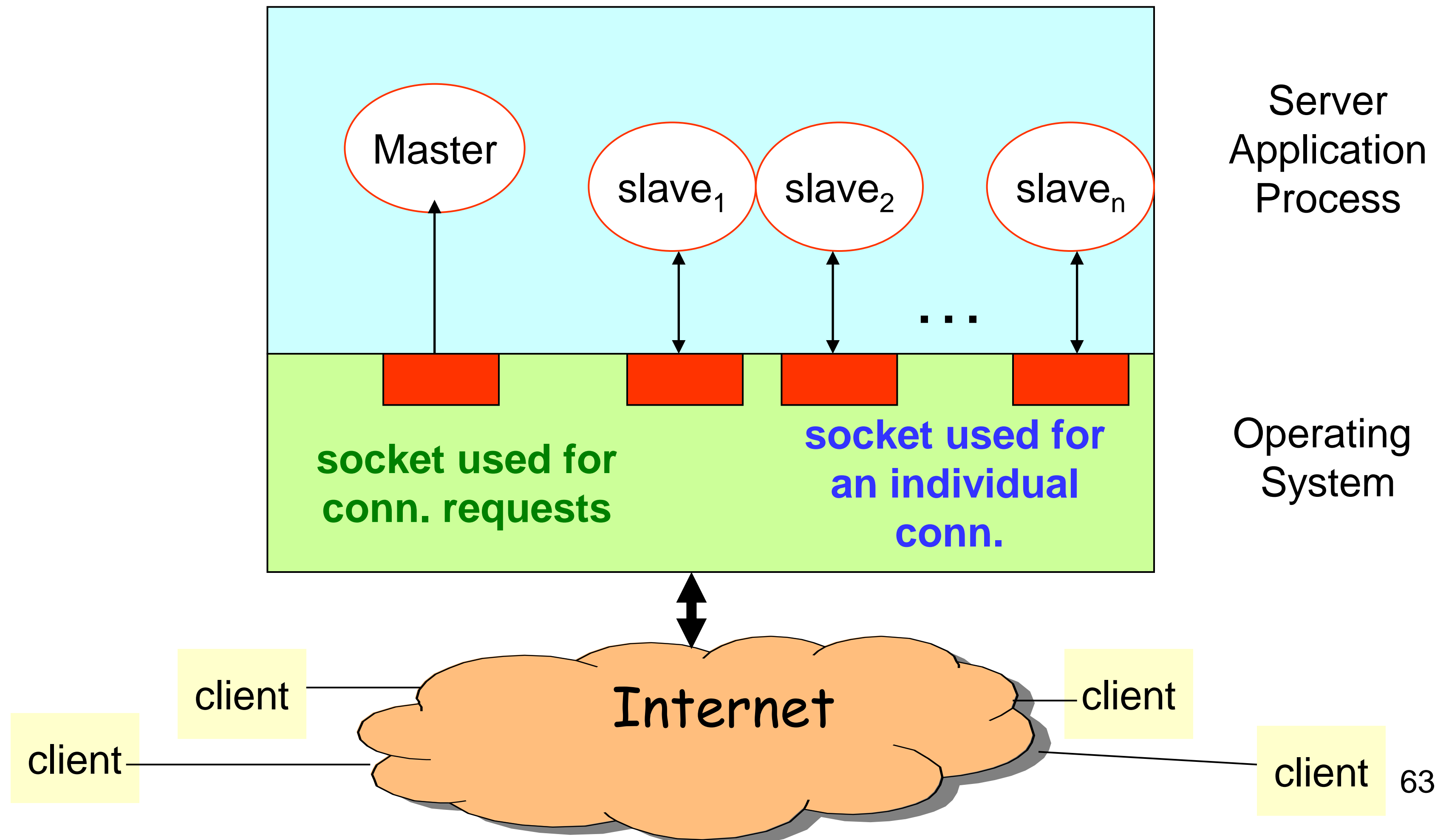
Concurrent sever: Using separate process

- A master server process begins execution initially
- Master opens a socket at the well-known port
- Wait for the next request
- Create a slave server process to handle each request
- A slave process exits when complete the communication with the client

- A concurrent server creates a new process for each connection.
- **fork()** – a system call
- Both master and slave processes execute the same code.
- execve() – have the slave process execute an independently written code after the call to fork.

```
#include <unistd.h>
int execve(const char *filename, char *const argv[], char *const envp[]);  // executes program pointed to by filename.
```

The Process Structure



Asynchronous Socket Programming (1/3)

- Synchronous: handle one request at a time, each in turn
  - pros: simple
  - cons: any one request can hold up all the other request
- Asynchronous/Event-driven programming
- Fork : start a new process to handle each request
  - pros: easy
  - does *not* scale well, hundreds of connections means hundreds of process

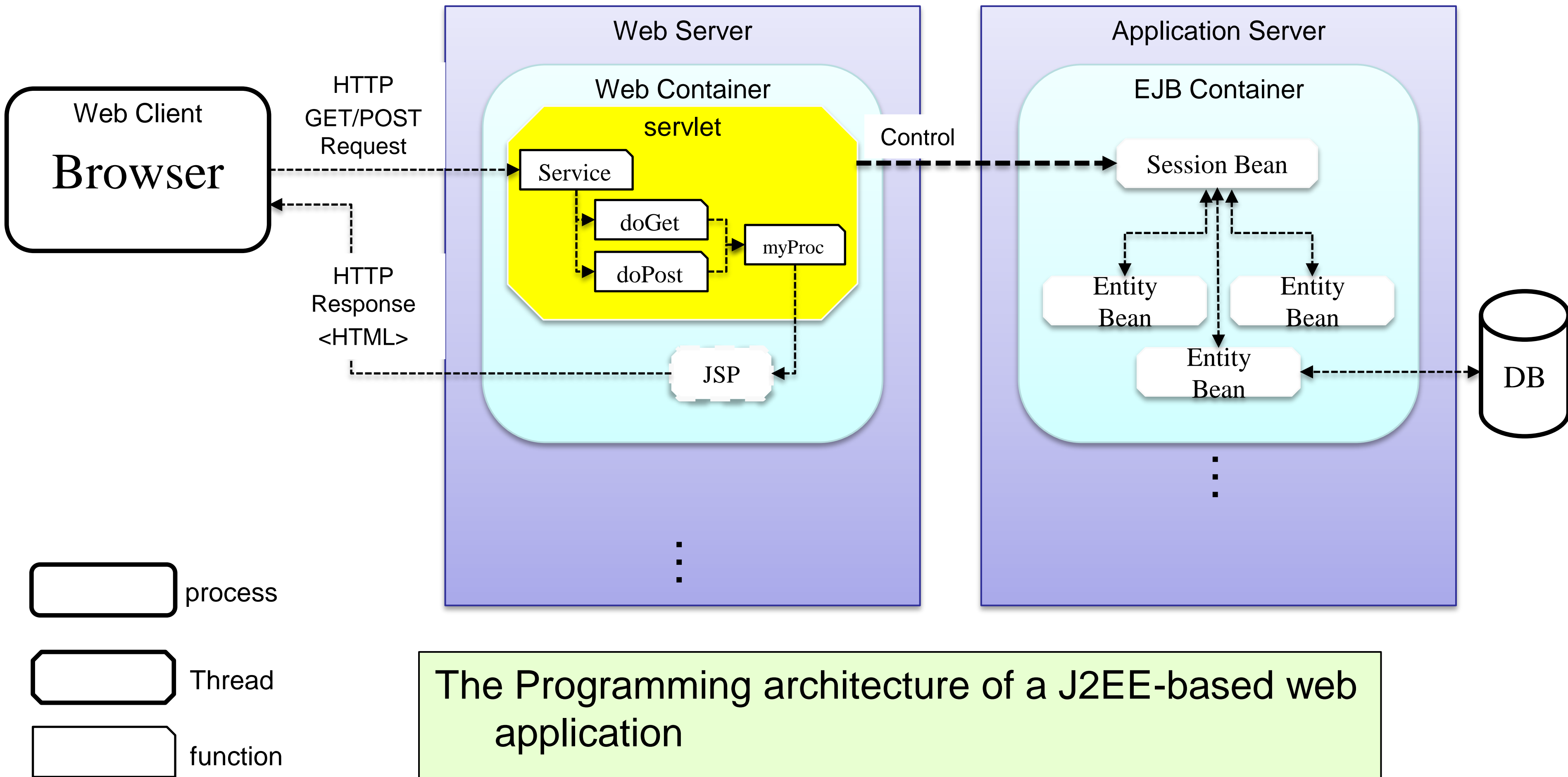
Concurrent Connection-Oriented Server Algorithm: using separate process

- Master
  - step 1: Create a socket and bind to the well-known address for the service being offered
  - step 2: Place the socket in passive mode, making it ready for use by a server
  - step 3: Repeatedly call *accept* to receive the next connection request from the client, and create a new slave process to handle the response.
- Slave
  - step 1: receive a connection request upon creation
  - step 2: interact with the client using the connection: read requests and send back replies
  - step 3: close the connection and exit.

Asynchronous Socket Programming (2/3)

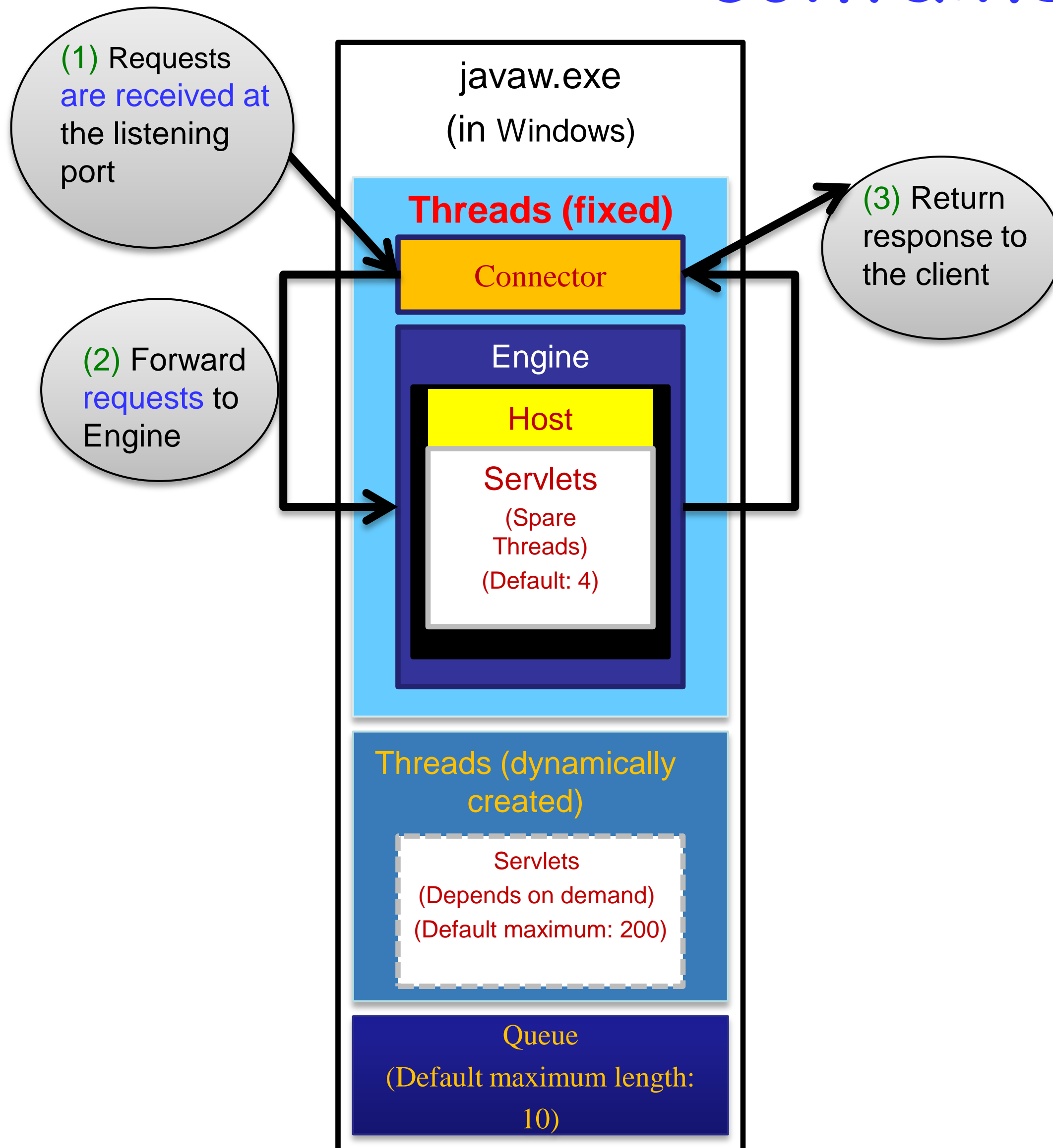
- In many OSs, process creation and context switching are expensive
- Threads: start a new thread to handle each request
  - pros:
    - easy
    - kinder to the kernel than using fork, since threads usually have much *less* overhead
  - cons:
    - local host machine needs to support threads;
    - threaded programming can get very complicated very fast, with worries about controlling access to shared resources

Concurrent sever: using thread worker pool



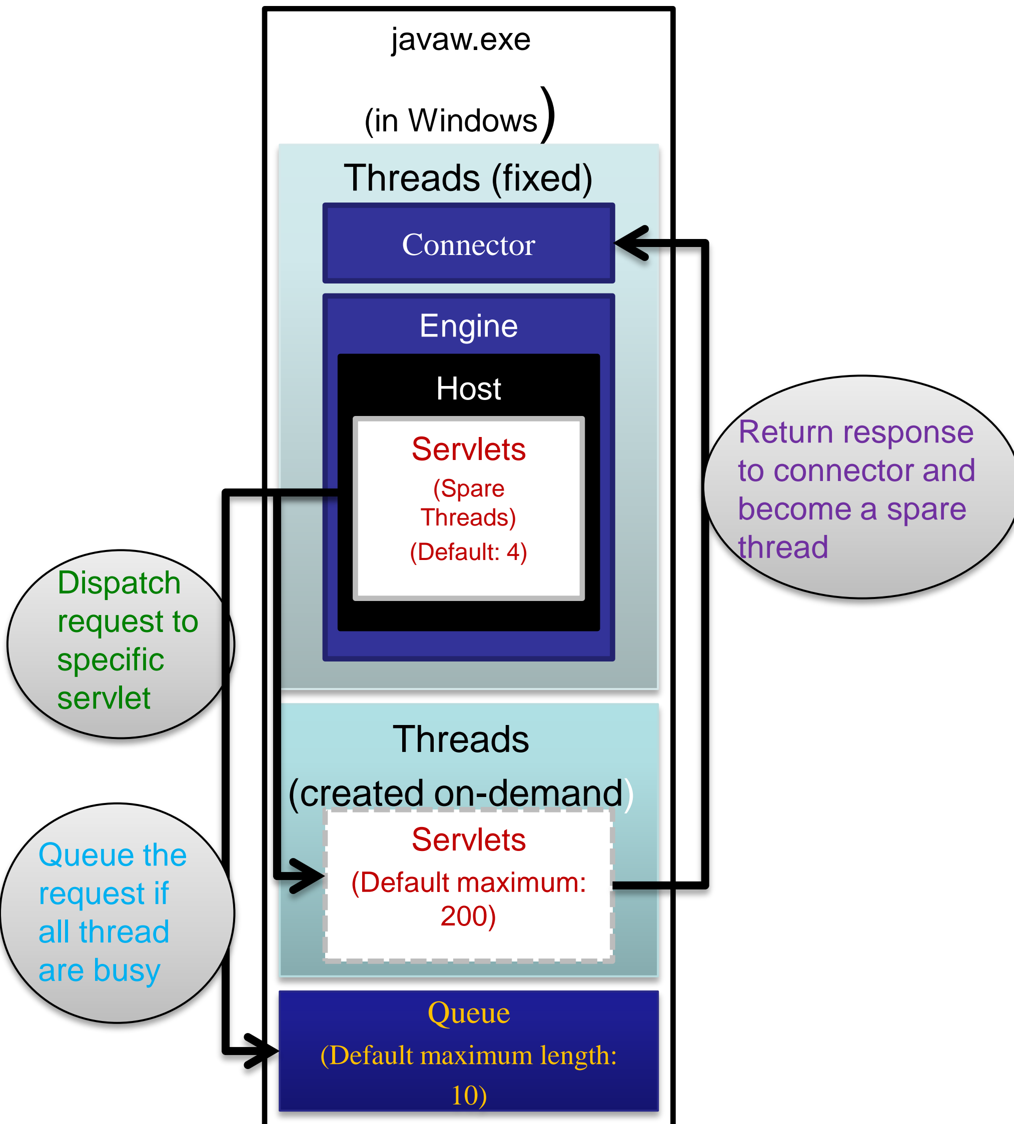
The Programming architecture of a J2EE-based web application

Internal structure of a typical Web container



- An active thread is designated as Connector to
  1. Receive requests from clients.
  2. Forward requests to Engine
  3. Return the results to the requesting client
- An active thread is designated as the ***host as the thread controller*** to
  1. Allocate an idle active thread in the thread pool to initialize specific servlet.
  2. Create additional thread to "compensate" Spare Thread if current spare thread is lower than minSpareThreads.
  3. If the total number of active Threads reaches the maxThread, then pend the request to Queue.

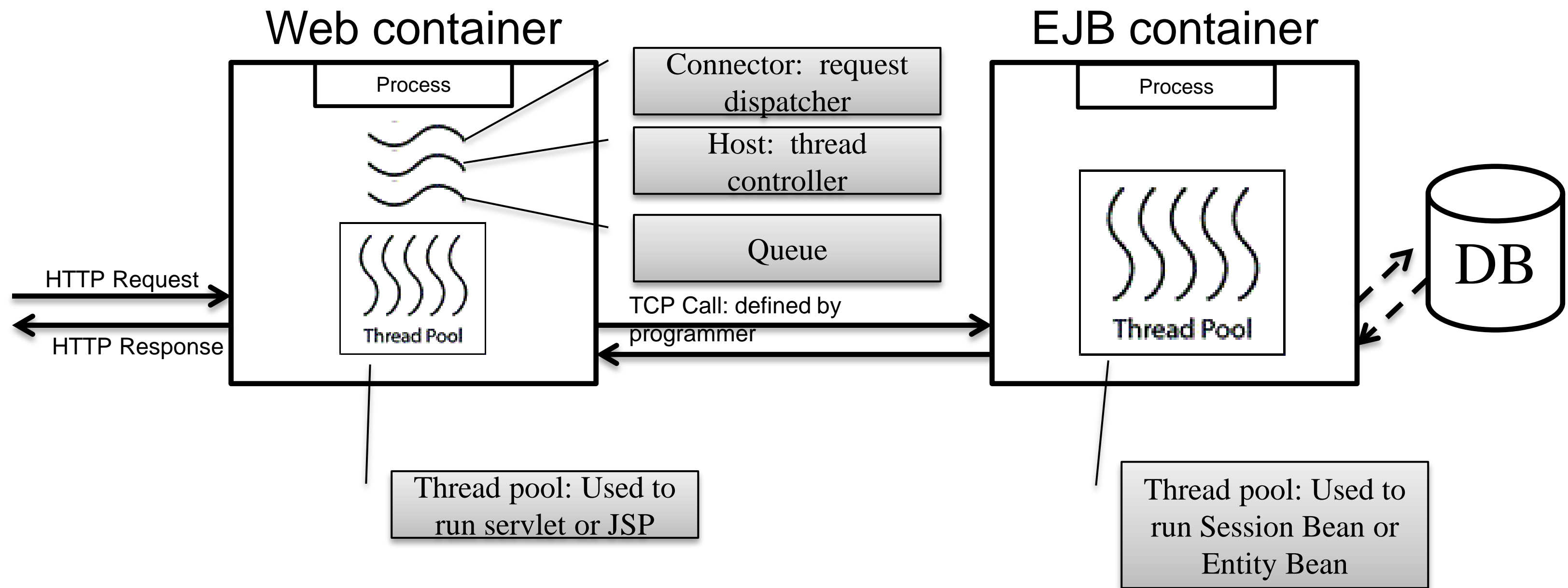
Web container - Host



- When servlet returns the response message, the execution terminates and the executing thread returns to the **pool** with the state marked as spare.
- If spare threads exceed maxSpareThreads, then host will destroy unnecessary spare Thread.
  - Attribute:
    - maxThreads
    - maxSpareThreads
    - minSpareThreads
  - Queue:
    - Function
      - Drop the request if queue is full.
    - Attribute:
      - Maximum queue length

Figure 5

Model of a typical J2EE web application



Asynchronous Socket Programming (3/3)

- select()/poll() – based
  - Pros:
    - efficient and elegant
    - scale well
    - require no interlocking for access to shared resource
    - integrates easily with event-driven window-system programming.
  - Cons:
    - more complex
    - require a fundamentally different approach to programming that can be confusing at first

Passive TCP: sample code (1/7)

```
/* passiveTCP.c - passiveTCP - create a passive socket for
   use in a TCP server */

int passivesock(const char *service, const char *transport,
   int qlen);

int passiveTCP(const char *service, int qlen)

/*      service - service associated with the desired port
 *      qlen    - maximum server request queue length */
{
   return passivesock(service, "tcp", qlen);
}
```

Passive TCP: sample code (2/7)

```
/* passivesock.c - passivesock */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>

extern int    errno;
int  errexit(const char *format, ...);
u_short  portbase = 0; // port base, for non-root servers
```

Passive TCP: sample code (3/7)

```
/*-------------------------------------------------------*
    passivesock –
    allocate & bind a server socket using TCP or UDP
 *-------------------------------------------------------*/
int passivesock(const char *service,
                const char *transport, int qlen)
/*
 * Arguments:
 *      service   - service associated with the desired port
 *      transport - transport protocol to use ("tcp" or "udp")
 *      qlen      - maximum server request queue length
 */
```

Passive TCP: sample code (4/7)

```
{
    struct servent          *pse;
            /* pointer to service information entry */
    struct protoent *ppe;
            /* pointer to protocol information entry */
    struct sockaddr_in sin;
            /* an Internet endpoint address */
    int   s, type;
            /* socket descriptor and socket type */

1.  memset(&sin, 0, sizeof(sin));
2.  sin.sin_family = AF_INET;
3.  sin.sin_addr.s_addr = INADDR_ANY;
```

Passive TCP: sample code (5/7)

```
/* Map service name to port number */
4. if ( pse = getservbyname(service, transport) )
5.          sin.sin_port = htons(ntohs((u_short)pse->s_port)
                + portbase);

6. else if ( (sin.sin_port = htons((u_short)atoi(service)))
                == 0 )
7.       errexit("can't get \"%s\" service entry\n", service);
/* Map protocol name to protocol number */
8.   if ( (ppe = getprotobyname(transport)) == 0)
9.       errexit("can't get \"%s\" protocol entry\n",  transport);
```

Passive TCP: sample code (6/7)

```
/* Use protocol to choose a socket type */
10.     if (strcmp(transport, "udp") == 0)
11.             type = SOCK_DGRAM;
12.    else   type = SOCK_STREAM;

/* Allocate a socket */
13.    s = socket(PF_INET, type, ppe->p_proto);
14.    if (s < 0)
         errexit("can't create socket: %s\n", strerror(errno));
```

Passive TCP: sample code (7/7)

```
/* Bind the socket */
15.  if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
         errexit("can't bind to %s port: %s\n", service,
                 strerror(errno));

16.  if (type == SOCK_STREAM && listen(s, qlen) < 0)
         errexit("can't listen on %s port: %s\n", service,
                 strerror(errno));
17.  return s;
}
```

Appendix

Generic address structure (1/3)

- **Generic address structure**
  - **The goal is to allow a software to manipulate protocol addresses without knowing the details of how every protocol family defines its address representation.**
  - e.g.,
    - a procedure that accepts an arbitrary protocol endpoint spec as an argument
    - Choose one of several possible actions depending on the address type
  - (address family, endpoint address in the family)
    - A constant denotes a predefined address types
    - The representation for the specified address type.

Generic address structure (2/3)

- To keep programs portable and maintainable, TCP/IP code should not use the sockaddr structure in declarations
- It can be used only as an overlay.
- The code should reference only the sa_family field.

Generic address structure (3/3)

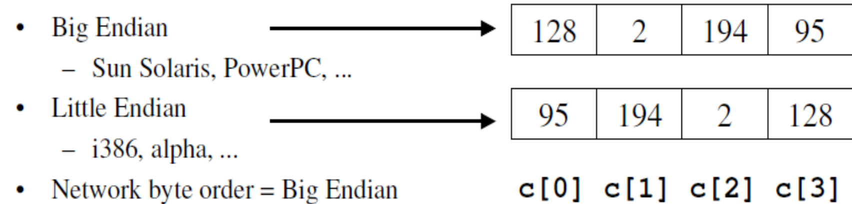
```
/* used for declaring variables to store endpoint
   address

struct sockaddr {          /* struct to hold an address  */
   u_char      sa_len;          /* total length               */
   u_short     sa_family;       /* type of address            */
   char        sa_data[14]; /* value of address           */
};
```

Network byte order

- Networks generally use big-endian order
- The Internet Protocol defines a standard big-endian *network byte order* for sending information over a network in a common format.
- This byte order is used for all *numeric* values in the packet headers and by many higher level protocols and file formats that are designed for use over IP.

Network byte order

- The Berkeley sockets API defines a set of functions to convert 16- and 32-bit integers to and from network byte order:
- from machine (*host*) to network order
  - htonl (host-to-network-long) (32-bit)
  - htons (host-to-network-short) (16-bit)
- from network to host order.
  - ntohl (32-bit)
  - ntohs (16-bit)



- All the layers above them usually consider *byte* (*octet*) as the atomic unit.