
Topic 5: Sockets*

K24: Systems Programming

Instructor: Mema Roussopoulou

Cross-host Interprocess Communication (IPC)

- ◆ We've seen (or will see) IPC on the same machine via
 - ◆ files
 - ◆ signals
 - ◆ pipes,
 - ◆ shared memory
 - ◆ ...
- ◆ What happens if a computer network lies between the processes?

Cross-host Interprocess Communication (IPC)

- ◆ Typically client-server model over network
- ◆ Server – provides a service
- ◆ Server – waits for client to connect
- ◆ Clients – connect to utilize the service
- ◆ Clients – possibly more than one at a time

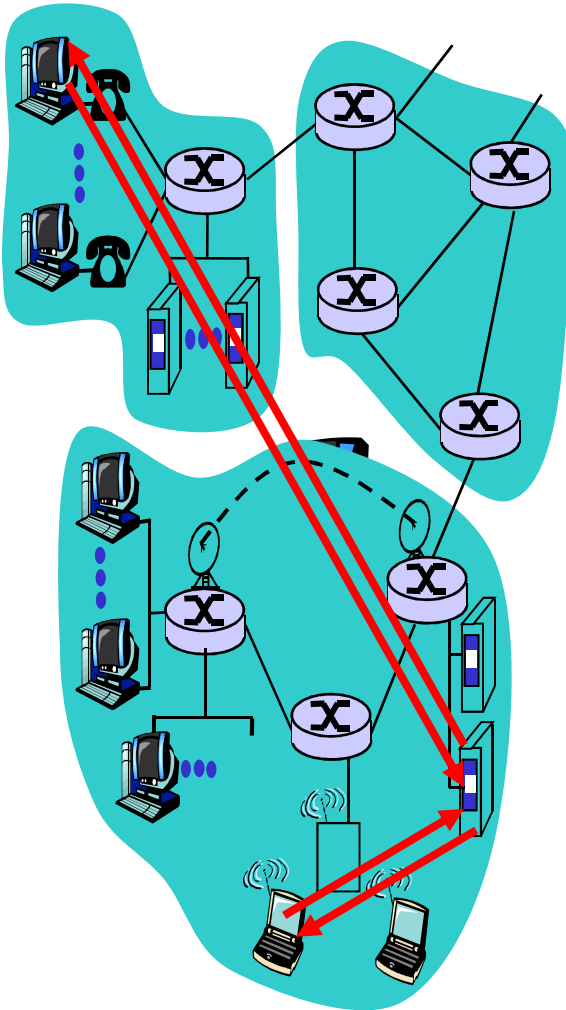
Some networked applications/services

- E-mail
- Web
- Instant messaging
- Remote login
- P2P file sharing
- Multi-user network games
- Streaming stored video clips
- Internet telephone
- Real-time video conference
- Massive parallel computing
-
-
-

Communication Models

- ◆ Client-server
- ◆ Peer-to-peer (P2P)
- ◆ Hybrid of client-server and P2P

Client-server architecture



server:

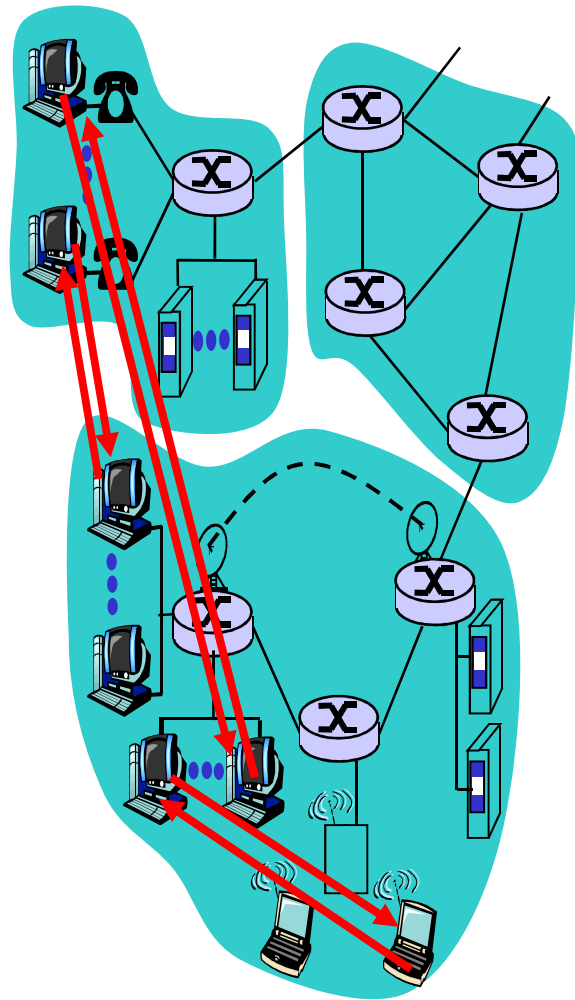
- always-on host
- permanent IP address
- server farms for scaling

clients:

- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

Pure P2P architecture

- no always-on server
- arbitrary end systems directly communicate
- each node (peer) acts as both a server and a client
- peers are intermittently connected and change IP addresses
- example: Gnutella



Highly scalable but
difficult to manage

Hybrid of client-server and P2P

Skype

- Internet telephony app
- Finding address of remote party: centralized server(s)
- Client-client connection is direct (not through server)

Instant messaging

- Chatting between two users is P2P
- Presence detection/location centralized:
 - User registers IP address with central server when she goes online
 - User contacts central server to find IP addresses of buddies

The Internet Protocol

- ◆ Each device in a network is assigned an IP address
- ◆ IPv4 32 bit, IPv6 128 bit
 - IPv4 (in decimal)
69.89.31.226 ← 4 octets
 - IPv6 (in hex)
2001:0db8:0a0b:12f0:0000:0000:0000:0001
← 8 16 bit blocks
- ◆ Each device may host many services
- ◆ Accessing a service requires an (IP, port) pair
- ◆ Services you know of: ssh (port 22), http (port 80), DNS (port 53), DHCP (ports 67,68)

Common service use cases

- ◆ **Browse the World Wide Web**
 - Each device has a static IP
 - DNS used to translate www.google.com to 216.58.213.4
 - Contact service at 216.58.213.4 and port 80 (http)

Common service use cases

◆ Your home network

- You turn on your modem. It gets a public IP address from your ISP (e.g., 79.166.80.131)
- Your modem runs a DHCP server giving IPs in 192.168.x.y
- Your modem acts as an Internet gateway. Translates IPs from 192.168.x.y to 79.166.80.131. (NAT box, IP Masquerade)
- What if you need to set up a service running inside your 192.168.x.y network available to the internet? Do port forwarding.

Processes communicating

Process: program running within a host.

- within same host, two processes communicate using **inter-process communication** (defined by OS).
- processes in different hosts communicate by exchanging **messages**

Client process:
process that initiates communication

Server process:
process that waits to be contacted

- Note: applications with P2P architectures have client processes & server processes

Transport Services Offered by OS

- ◆ OS deals with host to host data transfer
 - Did the data get there?
 - What to do when the data gets there?
- ◆ Supplies the application layer with a socket API (connect, send, receive)
- ◆ When data arrives at a host from the net, OS determines which application process is to receive the data (via socket)
- ◆ Applications typically use one of
 - Transmission Control Protocol (TCP)
 - User Datagram Protocol (UDP)

What transport service does an application need?

Data loss

- some apps (e.g., audio) can tolerate some loss
- other apps (e.g., file transfer, telnet) require 100% reliable data transfer

Timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

Bandwidth

- some apps (e.g., multimedia) require minimum amount of bandwidth to be “effective”
- other apps (“elastic apps”) make use of whatever bandwidth they get

OS Transport Services (TCP and UDP)

TCP service:

- *connection-oriented*: setup required between client and server processes
- *reliable transport* between sending and receiving process
- *flow control*: sender won't overwhelm receiver
- *congestion control*: throttle sender when network overloaded
- *does not provide*: timing, minimum bandwidth guarantees

UDP service:

- unreliable data transfer between sending and receiving process
- does not provide: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee

Q: why bother?
Why is there a UDP?

TCP versus UDP

Applications using TCP:

- HTTP (WWW), FTP (file transfer), Telnet (remote login), SMTP (email)

Applications using UDP:

- streaming media, teleconferencing, DNS, Internet telephony

How would you choose between TCP and UDP?

Processes communicating

Process: program running within a host.

- within same host, two processes communicate using **inter-process communication** (defined by OS).
- processes in different hosts communicate by exchanging **messages**

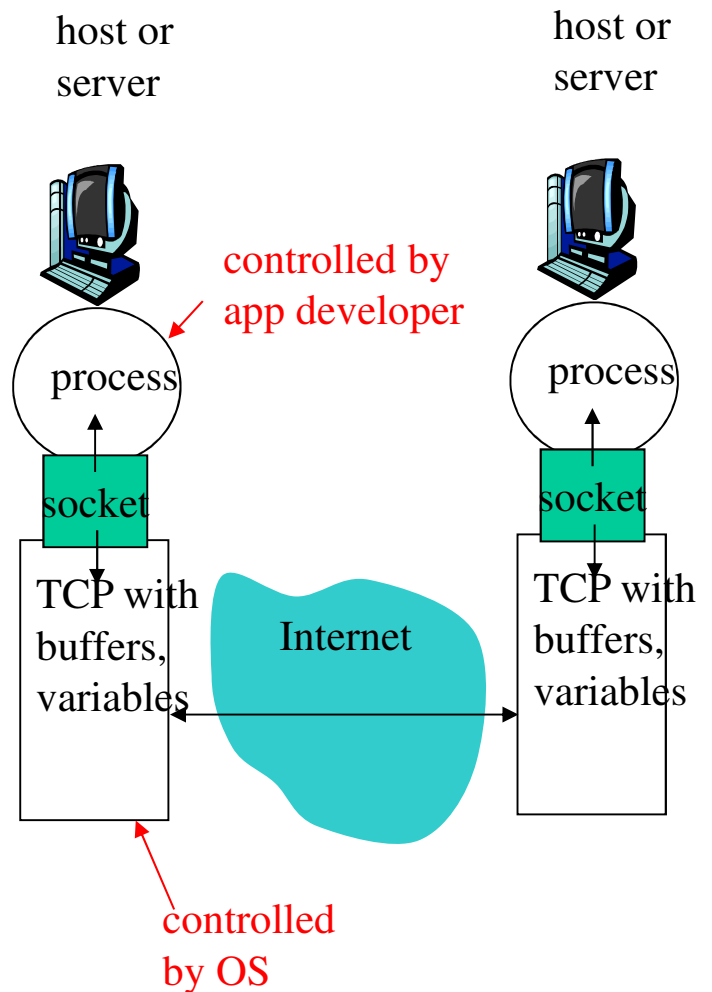
Client process:
process that initiates communication

Server process:
process that waits to be contacted

- Note: applications with P2P architectures have client processes & server processes

Sockets (Υποδοχές)

- process sends/receives messages to/from its **socket**
- socket analogous to door
 - sending process shoves message out door
 - sending process relies on transport infrastructure on other side of door which brings message to socket at receiving process
- API: (1) choice of transport service OS will use; (2) ability to fix a few parameters



Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- **Q:** does IP address of host on which process runs suffice for identifying the process?

Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- **Q:** does IP address of host on which process runs suffice for identifying the process?
 - **Answer:** NO, many processes can be running on same host
- *identifier* includes both **IP address** and **port numbers** associated with process on host.
- Example port numbers:
 - HTTP server: 80
 - Mail server: 25
- to send HTTP message to www.di.uoa.gr web server:
 - IP address: 147.52.17.2
 - Port number: 80

Application protocol defines

- Types of messages exchanged,
 - e.g., request, response
 - Message syntax:
 - what fields in messages & how fields are delineated
 - Message semantics
 - meaning of information in fields
 - Rules for when and how processes send & respond to messages
- Public-domain protocols:**
- defined in RFCs
 - allows for interoperability
 - e.g., HTTP, SMTP
- Proprietary protocols:**
- e.g., Skype, KaZaA, etc.

Socket programming

Goal: learn how to build client/server application programs that communicate using sockets

Socket API

- ◆ introduced in BSD4.1 UNIX, 1981
- ◆ explicitly created, used, released by applications
- ◆ client/server paradigm
- ◆ two types of transport service via socket API:
 - unreliable datagram (UDP)
 - reliable, byte stream-oriented (TCP)

socket

a *host-local, application-created, OS-controlled* interface (a "door") into which application process can **both send and receive** messages to/from another application process

More on sockets

- ◆ A socket is a communication endpoint
- ◆ Processes refer to a socket using an integer (file) descriptor
- ◆ Communication domain
 - Internet domain (over internet)
 - Unix domain (same host)
- ◆ Communication type
 - Stream (usually TCP)
 - Datagram (usually UDP)

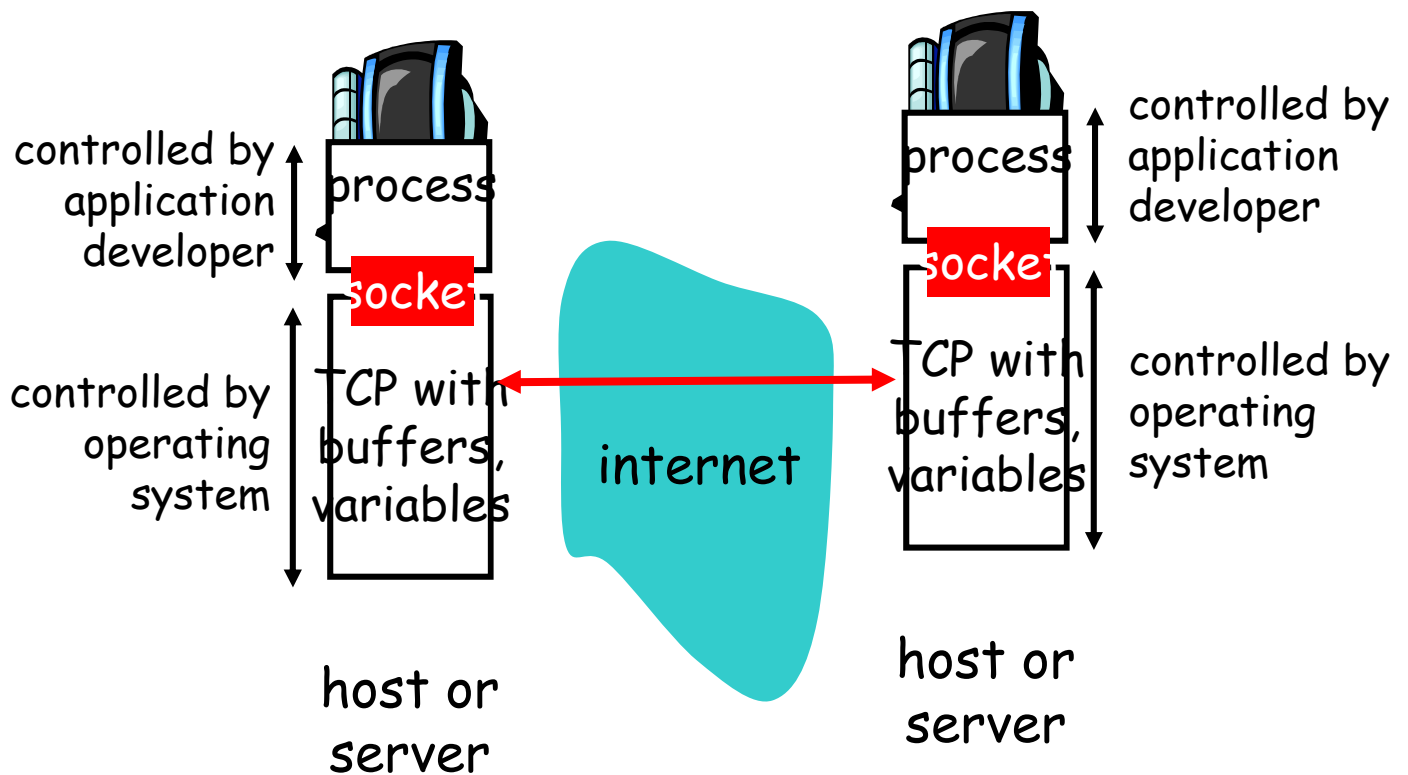
TCP versus UDP

	TCP	UDP
CONNECTION REQUIRED	YES	NO
RELIABILITY	YES	NO
MESSAGE BOUNDARIES	NO	YES
IN-ORDER DATA DELIVERY	YES	NO
SOCKET TYPE	SOCK_STREAM	SOCK_DGRAM

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



Socket programming with TCP

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

- creating client-local TCP socket
- specifying IP address, port number of server process
- When **client creates socket**: client TCP establishes connection to server TCP

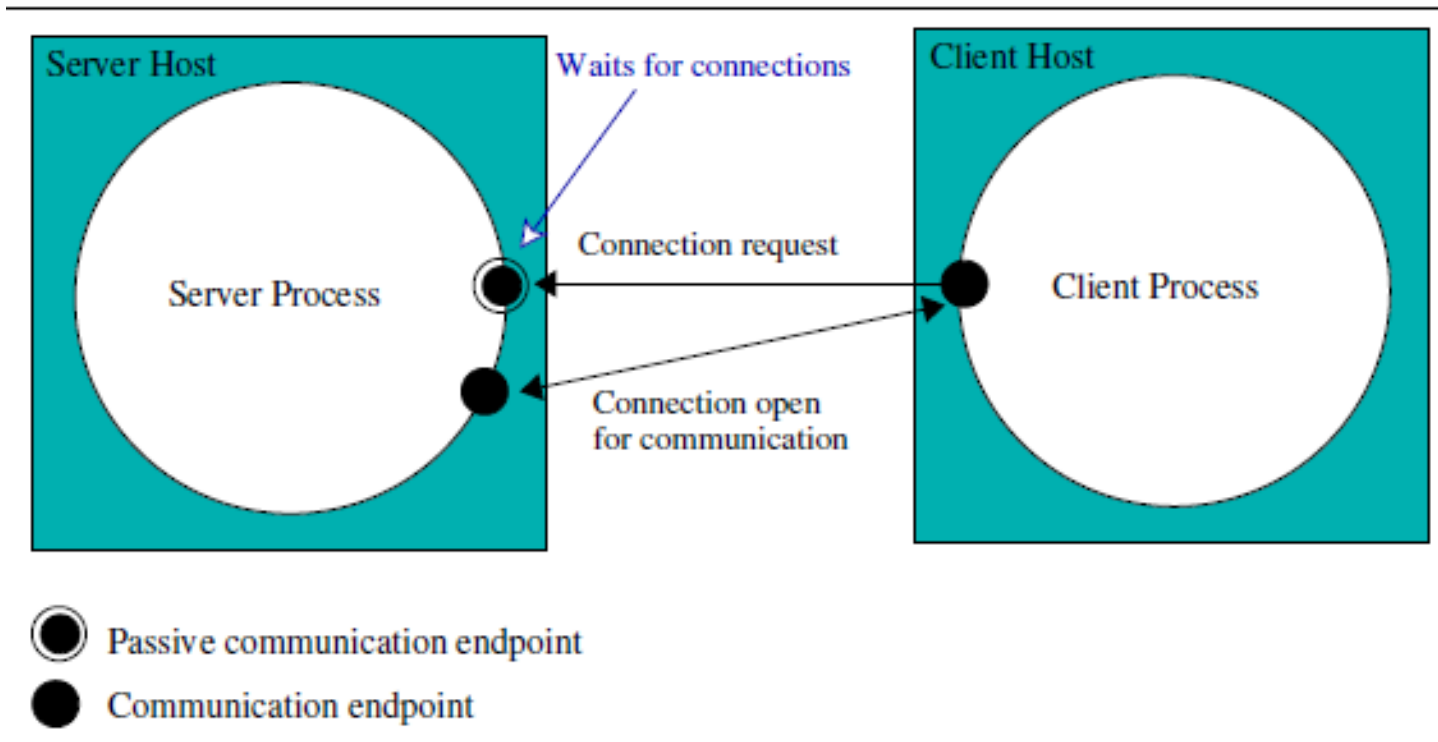
- When contacted by client, **server TCP creates new socket** for server process to communicate with client
 - allows server to talk with multiple clients
 - source port numbers used to distinguish clients

application viewpoint

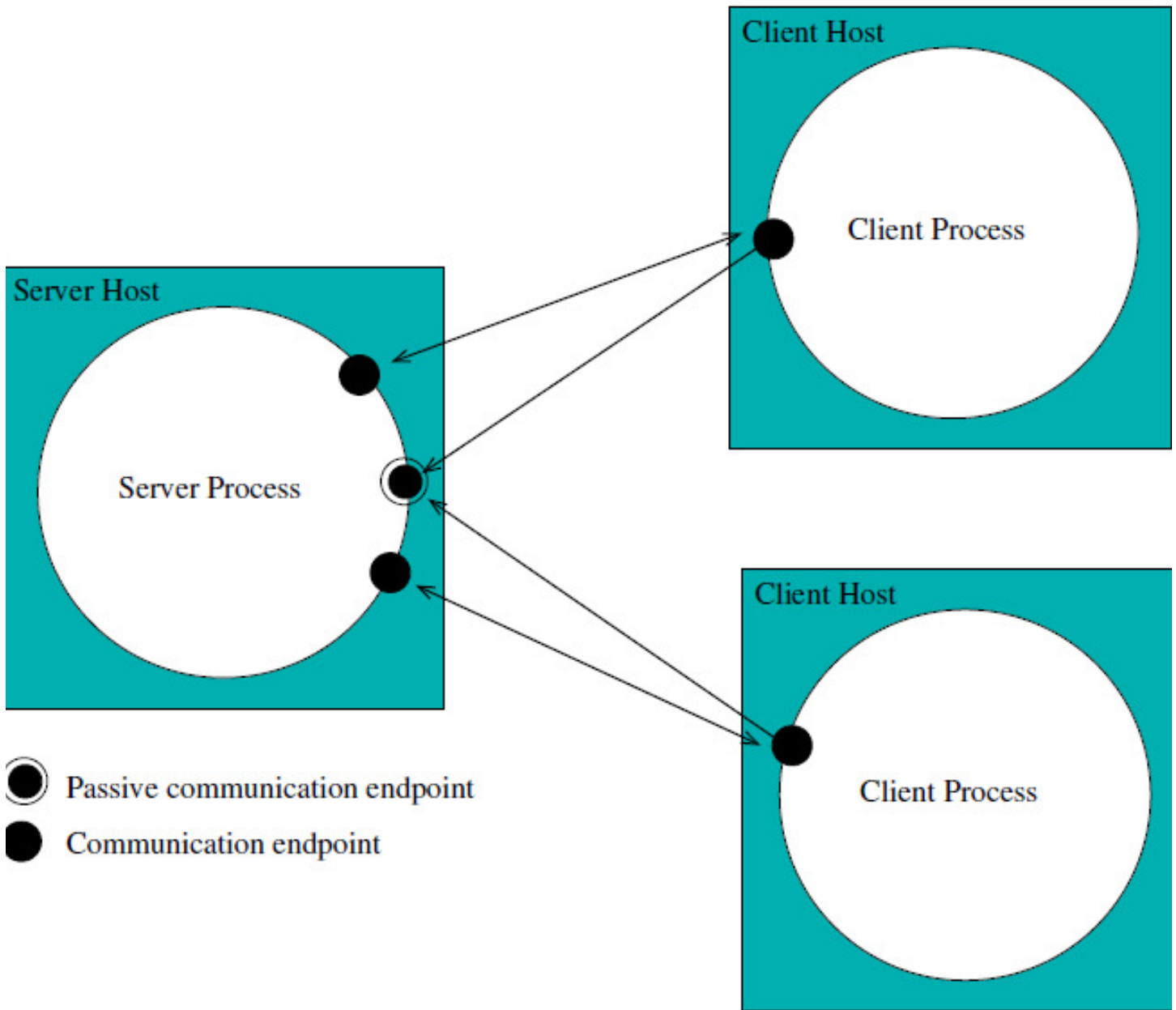
TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

More on TCP

- ◆ TCP uses acknowledgements
- ◆ Non-acknowledged messages are retransmitted
- ◆ Messages re-ordered by the receiver's OS network stack
- ◆ Application thus sees a properly ordered *data stream*



TCP – multiple clients



Serial server (TCP)

```
for (;;) {  
    wait for a client request on the listening file descriptor  
    create a private two-way communication channel to the client  
    while (no error on the private communication channel)  
        read from the client  
        process the request  
        respond to the client  
    close the file descriptor for the private communication channel  
}
```

ALWAYS



DRAWBACKS?

Serial server (TCP)

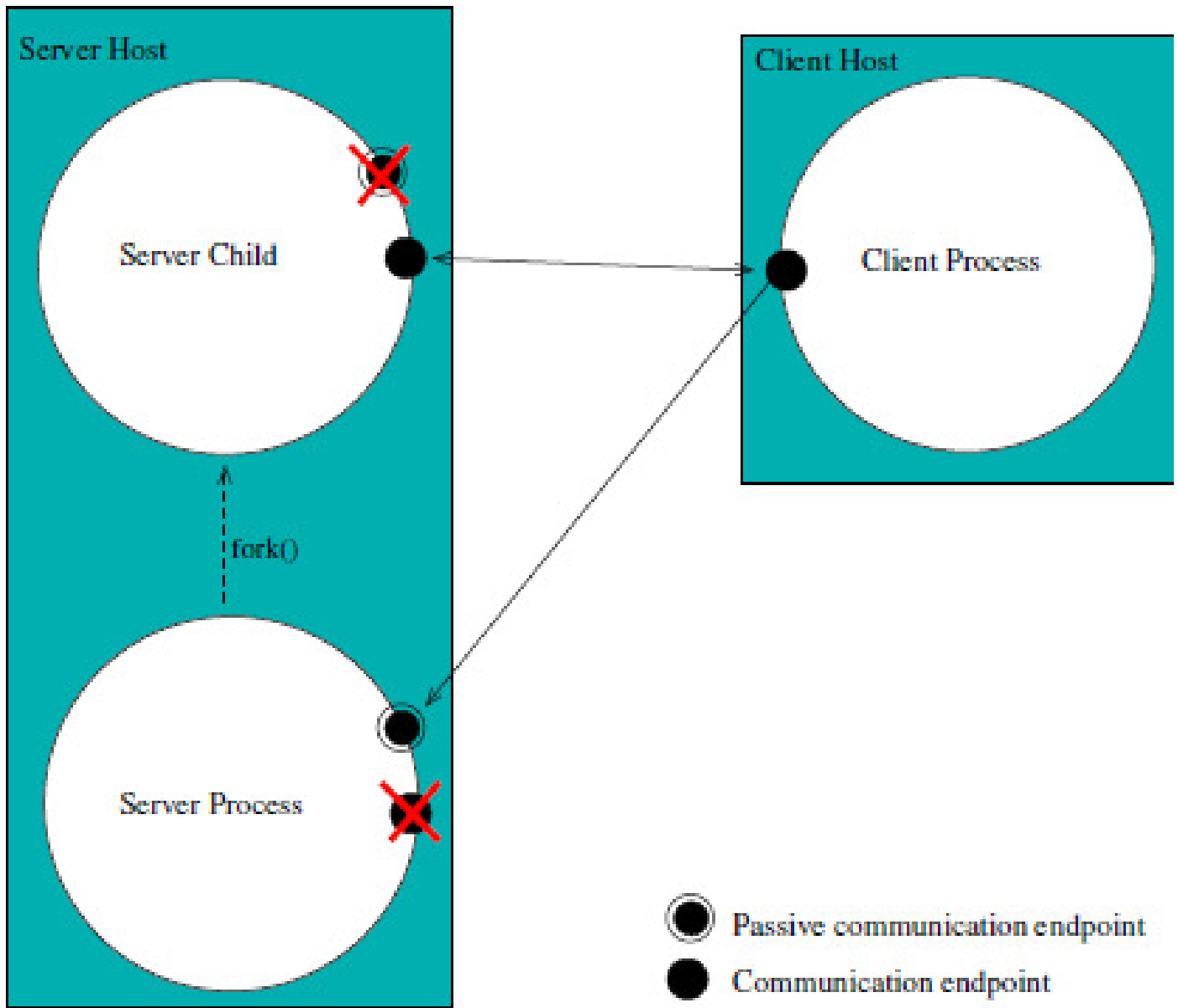
```
for (;;) {  
    wait for a client request on the listening file descriptor  
    create a private two-way communication channel to the client  
    while (no error on the private communication channel)  
        read from the client  
        process the request  
        respond to the client  
    close the file descriptor for the private communication channel  
}
```

 **ALWAYS**

DRAWBACKS

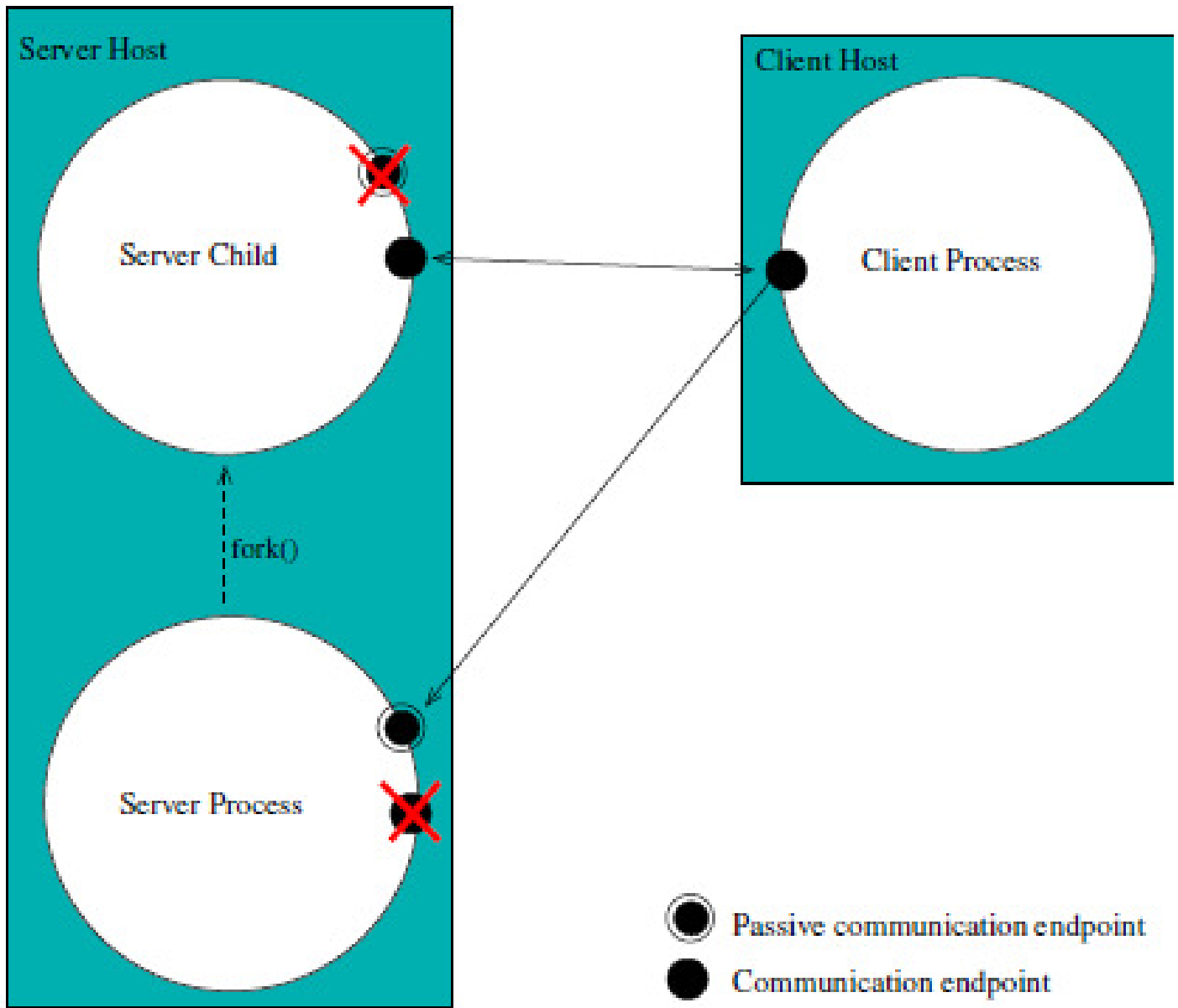
- 1) Serves only one client at a time
- 2) Other clients are forced to wait or even fail

1 process per client model



- ◆ New process forked for each client
- ◆ Multiple clients served at the same time
- ◆ Drawbacks?

1 process per client model



- ◆ New process forked for each client
- ◆ Multiple clients served at the same time
- ◆ Inefficient, too many clients → too many processes

1 process per client model

Parent process

```
for (;;) {  
    wait for a client request on the listening file descriptor  
    create a private two-way communication channel to the client  
    fork a child to handle the client  
    close the file descriptor for the private communication channel  
    clean up zombie children  
}
```

ALWAYS

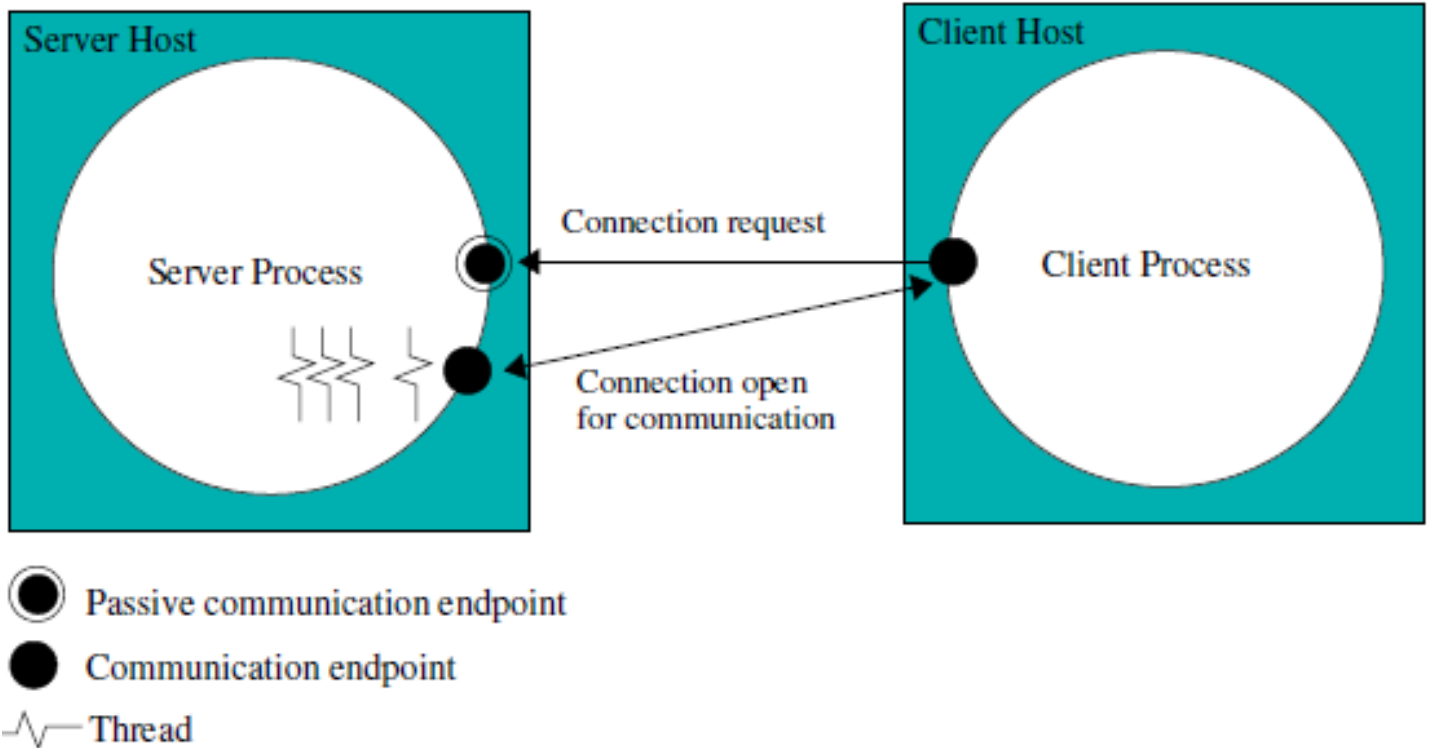
Child process

```
close the listening file descriptor  
handle the client  
close the communication for the private channel  
exit
```

Why should parent process close file descriptor for the private channel?

- ◆ Avoids risk of running about of file descriptors
- ◆ Enables the destruction of the channel once the other two parties (child & client) close their file descriptors
- ◆ Enables the child process to receive EOF after the client closes its end of the channel (and vice versa)

Multithreaded server model



- ◆ Create a new thread for each client
- ◆ Multiple threads handle multiple clients concurrently
- ◆ Drawback: requires thread synchronization for access to shared resources

Dealing with byte order

- Ένας ακέραιος με δεκαεξαδική αναπαράσταση **0xD04C** σε little-endian αρχιτεκτονική θα ερμηνευτεί ως **0x4CD0** σε big-endian host.
(π.χ., SUN (big-endian) vs Intel (little-endian))
- Η αποστολή δυαδικών αριθμών πρέπει να γίνεται με κάποια προσυμφωνημένη δικτυακή διάταξη (**Network Byte Order**)
- Συναρτήσεις βιβλιοθήκης htons, htonl, ntohs, και ntohs
unsigned short htons(unsigned short hostshort)
unsigned long htonl(unsigned long hostlong)
unsigned short ntohs(unsigned short netshort)
unsigned long ntohl(unsigned long netlong)
- Μετατροπή ακολουθίας bytes από διάταξη «μηχανής» σε διάταξη «δικτύου» και αντίστροφα για short και long ακεραίους
- Απαιτήσεις
#include <sys/types.h>
#include <netinet/in.h>

From *Domain Names* to *Addresses* and back

- Συναρτήσεις βιβλιοθήκης `gethostbyname` και `gethostbyaddr`
 - `struct hostent *gethostbyname(char *name)`
 - `struct hostent *gethostbyaddr(char *addr, int len, int type)`
- Η `gethostbyname` βρίσκει τη διεύθυνση IP που αντιστοιχεί στο όνομα `name`.
- Επιστρέφει έναν δείκτη σε δομή `struct hostent` όπου στο πεδίο `h_addr` της δομής βρίσκεται η Internet διεύθυνση και στο πεδίο `h_length` το μέγεθος της
- Η `gethostbyaddr` μετατρέπει μια διεύθυνση IP `addr`, με μέγεθος `len`, και τύπου `type` (πάντα `AF_INET`), σε όνομα.
- Επιστρέφει έναν δείκτη σε δομή `struct hostent` όπου στο πεδίο `h_name` βρίσκεται το όνομα του υπολογιστή
- Επιστρέφουν `NULL` σε περίπτωση σφάλματος (ορίζει το σφάλμα το `h_errno`)
- Απαιτήση: `#include <netdb.h>`

socket()

• Όλες οι κλήσεις συστήματος που ακολουθούν και χρησιμοποιούνται για διαχείριση υποδοχών απαιτούν

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

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

και επιστρέφουν -1 σε περίπτωση αποτυχίας

• Κλήση συστήματος socket

```
int socket(int domain, int type, int protocol)
```

Δημιουργεί μια υποδοχή και επιστρέφει ένα περιγραφέα αρχείου που αντιστοιχεί σ' αυτήν

Το **domain** πρέπει να είναι **AF_INET** (παλιά **PF_INET**)

Το **type** πρέπει να είναι **SOCK_STREAM** ή
SOCK_DGRAM

Σαν **protocol**, πάντα δίνουμε το default (0)

```
int sock;
```

```
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)  
    perror("Failed to create socket");
```

TCP Communication

Server

socket ()



bind ()



listen ()



accept ()

αναμονη συνδεσης



read ()



επεξεργασία αιτησης



write ()

Client

socket ()



connect ()



write ()

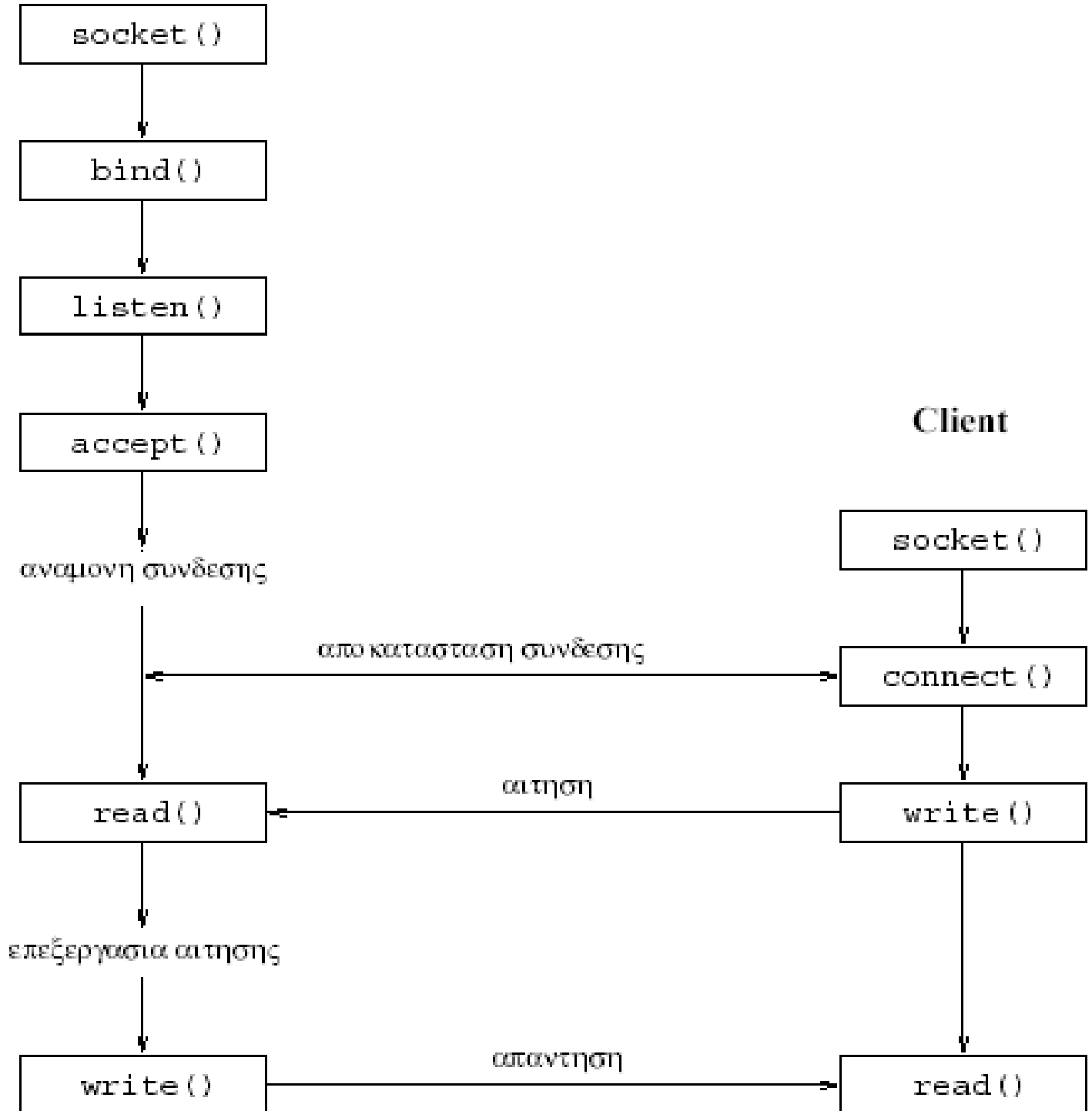


read ()

απο κατασταση συνδεσης

αιτηση

απαντηση



Address Format for Sockets

- ◆ An address identifies a socket in a specific communication domain.
- ◆ Addresses with different formats can be passed to the socket programming functions – all casted to the generic sockaddr structure.
- ◆ Internet addresses are defined in `<netinet/in.h>`.

Address Format for Sockets

- ◆ In IPv4 Internet domain (AF_INET), a socket address is represented by the sockaddr in as:

```
#include <netinet/in.h>
struct sockaddr_in {
    sa_family_t sin_family;    /* AF_INET */
    in_port_t sin_port;       /* port number */
    struct in_addr sin_addr;   /* IPv4 address */
};

struct in_addr {
    in_addr_t s_addr;        /* IPv4 unsigned 32 bit int
*/
};
```

- ◆ in_port_t data type is uint16_t (defined in <stdint.h>)
- ◆ in_addr_t data type is uint32_t (defined in <stdint.h>)

Binding sockets to addresses with bind()

- ◆ bind requests for an address to be assigned to a socket
- ◆ You must bind a `SOCK_STREAM` socket to a local address before receiving TCP connections
- ◆ Need `#include<netinet/in.h>`
- ◆ `int bind (int socket, const struct sockaddr *address, socklen_t address_len);`
- ◆ We pass a `sockaddr_in` struct as the address that has at least the following members expressed in network byte-order:
 - – `sin_family`: address family is `AF_INET` in the Internet domain
 - – `sin_addr.s_addr`: address can be a specific IP or `INADDR_ANY`
 - – `sin_port`: TCP or UDP port number

Socket binding example

```
#include <netinet/in.h> /* for sockaddr_in */
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h> /* for htonl * */

int bind_on_port ( int sock, short port ) {
    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl (INADDR_ANY);
    server.sin_port = htons(port);
    return bind(sock, (struct sockaddr *) &server,
                sizeof (server));
}
```

- ◆ INADDR ANY is a special address (0.0.0.0) meaning “any address”
- ◆ *sock* will receive connections from all addresses of the host machine

listen(), accept()

`int listen(int socket, int backlog);`

- ◆ Listen for connections **on a socket**
- ◆ **At most backlog connections will be queued waiting to be accepted**

`int accept(int socket, struct sockaddr *
address, socklen_t * address_len);`

- ◆ **Accepts a connection on a socket**
- ◆ **Blocks until a client connects/gets-interrupted by a signal**
- ◆ **Returns new socket descriptor to communicate with client**
- ◆ **Returns info on client's address through address. Pass NULL if you don't care.**
- ◆ **Value-result address len must be set to the amount of space pointed to by address (or NULL).**

connect()

```
int connect(int socket, struct sockaddr *address,  
            socklen_t address_len );
```

- ◆ **When called by a client**, a connection is attempted to a listening socket **on the server in address**. Normally, the server accepts the connection and a communication channel is established.
- ◆ If socket is of type **SOCK_DGRAM**, address specifies the peer with which the socket is to be associated (datagrams are sent/received only to/from this peer via the UDP protocol).

Library functions

bzero() and bcopy()

- ◆ void bzero(char *buf, int count)
 - Sets *count* bytes to 0 starting at address *buf*
- ◆ void bcopy(char *buf1, char *buf2, int count)
 - Copies *count* bytes starting from address *buf1* to address *buf2*
 - Opposite of memcpy
- ◆ Need #include<string.h>
- ◆ In Solaris compile with “-lsocket -lnsl”

Tips and warnings

- ◆ If a process attempts to write through a socket that has been closed by the other peer, a SIGPIPE signal is received.
- ◆ Since SIGPIPE is by default fatal, install a signal handler to override this.
- ◆ When a server quits, the listening port remains busy (state TIME WAIT) for a while
- ◆ Restarting the server *fails in bind* with “Bind: Address Already in Use”
 - To override this, use `setsockopt()` to enable `SO_REUSEADDR` before you call `bind()`.

TCP Επικοινωνία

Server

socket ()



bind ()



listen ()



accept ()



αναμονη συνδεσης



read ()



επεξεργασία αιτησης



write ()

Client

socket ()



connect ()



write ()

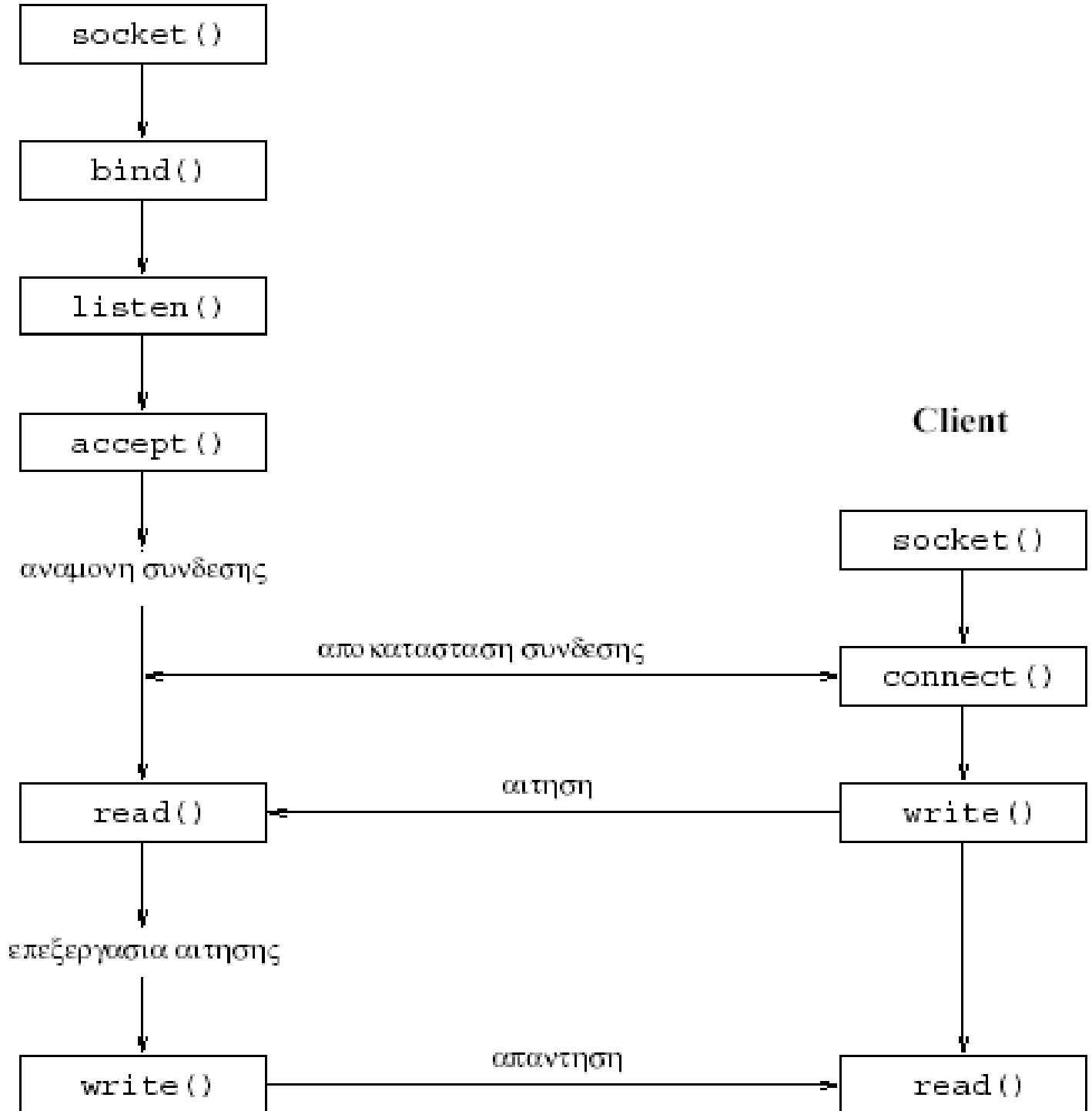


read ()

απο κατασταση συνδεσης

αιτηση

απαντηση



Example

- ◆ TCP server receives a string and replies with the string capitalized

```

/*inet_str_server.c: Internet stream sockets server */
#include <stdio.h>
#include <sys/wait.h>      /* sockets */
#include <sys/types.h>    /* sockets */
#include <sys/socket.h>   /* sockets */
#include <netinet/in.h>   /* internet sockets */
#include <netdb.h>        /* gethostbyaddr */
#include <unistd.h>       /* fork */
#include <stdlib.h>       /* exit */
#include <ctype.h>        /* toupper */
#include <signal.h>       /* signal */
void child_server(int newssock);
void perror_exit(char *message);
void sigchld_handler (int sig);

void main(int argc, char *argv[]) {
    int      port, sock, newssock;
    struct sockaddr_in server, client;
    socklen_t clientlen;
    struct sockaddr *serverptr=(struct sockaddr *)&server;
    struct sockaddr *clientptr=(struct sockaddr *)&client;
    struct hostent *rem;
    if (argc != 2) {
        printf("Please give port number\n");exit(1);}
    port = atoi(argv[1]);
    /* Reap dead children asynchronously */
    signal(SIGCHLD, sigchld_handler);

```

/* Create socket */

```
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    perror_exit("socket");
```

```
server.sin_family = AF_INET; /* Internet domain */
```

```
server.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
server.sin_port = htons(port); /* The given port */
```

/* Bind socket to address */

```
if (bind(sock, serverptr, sizeof(server)) < 0)
```

```
    perror_exit("bind");
```

/* Listen for connections */

```
if (listen(sock, 5) < 0) perror_exit("listen");
```

```
printf("Listening for connections to port %d\n", port);
```

```
while (1) { clientlen = sizeof(client);
```

/* accept connection */

```
if ((newsock = accept(sock, clientptr, &clientlen))
    < 0) perror_exit("accept");
```

/* Find client's name */

```
if ((rem = gethostbyaddr((char *) &client.sin_addr.s_addr,
    sizeof(client.sin_addr.s_addr), client.sin_family))
    == NULL) {
```

```
    perror_exit("gethostbyaddr"); exit(1);} 
```

```
printf("Accepted connection from %s\n", rem->h_name);
```

```
switch (fork()) { /* Create child for serving client */
```

```
case -1: /* Error */
```

```
    perror_exit("fork"); break;
```

```
case 0: /* Child process */
```

```
    close(sock); child_server(newsock);
```

```
    exit(0);
```

```
}
```

```
close(newsock); /* parent closes socket to client */
    }
}
```

```
void child_server(int newsock) {
    char buf[1];
    while(read(newsock, buf, 1) > 0) { /* Receive 1 char */
        putchar(buf[0]); /* Print received char */
        /* Capitalize character */
        buf[0] = toupper(buf[0]);
        /* Reply */
        if (write(newsock, buf, 1) < 0)
            perror_exit("write");
    }
    printf("Closing connection.\n");
    close(newsock); /* Close socket */
}
```

```
/* Wait for all dead child processes */
```

```
void sigchld_handler(int sig) {
    while (waitpid(-1, NULL, WNOHANG) > 0);
}
```

```
void perror_exit(char *message) {
    perror(message);
    exit(EXIT_FAILURE);
}
```

And now the client...

```
/* inet_str_client.c: Internet stream sockets client */  
#include <stdio.h>  
#include <sys/types.h>      /* sockets */  
#include <sys/socket.h>    /* sockets */  
#include <netinet/in.h>    /* internet sockets */  
#include <unistd.h>        /* read, write, close */  
#include <netdb.h>         /* gethostbyaddr */  
#include <stdlib.h>        /* exit */  
#include <string.h>        /* strlen */
```

```
void perror_exit(char *message);
```

```
void main(int argc, char *argv[]) {  
    int      port, sock, i;  
    char    buf[256];  
    struct sockaddr_in server;  
    struct sockaddr *serverptr = (struct sockaddr*)&server;  
    struct hostent *rem;  
    if (argc != 3) {  
        printf("Please give host name and port number\n");  
        exit(1);}  
    /* Create socket */  
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)  
        perror_exit("socket");  
    /* Find server address */  
    if ((rem = gethostbyname(argv[1])) == NULL) {  
        herror("gethostbyname"); exit(1);  
    }
```

```

port = atoi(argv[2]);  /* Convert port number to integer */
server.sin_family = AF_INET;  /* Internet domain */
memcpy(&server.sin_addr, rem->h_addr, rem->h_length);
server.sin_port = htons(port);  /* Server port */
/* Initiate connection */
if (connect(sock, serverptr, sizeof(server)) < 0)
    perror_exit("connect");
printf("Connecting to %s port %d\n", argv[1], port);
do {
    printf("Give input string: ");
    fgets(buf, sizeof(buf), stdin); /* Read from stdin */
    for(i=0; buf[i] != '\0'; i++) { /* For every char */
        /* Send i-th character */
        if (write(sock, buf + i, 1) < 0)
            perror_exit("write");
        /* receive i-th character transformed */
        if (read(sock, buf + i, 1) < 0)
            perror_exit("read");
    }
    printf("Received string: %s", buf);
} while (strcmp(buf, "END\n") != 0); /* Finish on "end" */
close(sock);  /* Close socket and exit */
}

```

```

void perror_exit(char *message)
{
    perror(message);
    exit(EXIT_FAILURE);
}

```

Server on linux02:

mema@linux02> ./server 9002

Listening for connections to port 9002

Accepted connection from linux03.di.uoa.gr

Hello world

EnD

Closing connection .

Client on linux03:

mema@linux03> ./client linux02.di.uoa.gr 9002

Connecting to linux02.di.uoa.gr port 9002

Give input string : Hello world

Received string : HELLO WORLD

Give input string : EnD

Received string : END

mema@browser>

More useful tips

- ◆ **INADDR_ANY (0.0.0.0)**
 - Means “listen on all network interfaces” on this host
- ◆ **port = 0**
 - Lets the OS choose a port
 - Use `getsockname()` after `bind` to find which port was given to you
- ◆ **int setsockopt()**
 - Sets certain properties of a socket
 - We’ll see example use later
- ◆ **int shutdown(int socket_fd, int how)**
 - Shuts down part of a full-duplex connection
 - Can be used to tell server that we have sent the whole request
 - `how = SHUT_RD, SHUT_WR, or SHUT_RDWR`
 - Example: HTTP client sends request, then `shutdown(fd, SHUT_WR)` to tell server “I am done sending the whole request, but can still receive.” Server detects EOF by receive of 0 bytes and can assume it has complete request.
 - At end, we close the socket with `close()` call

More useful functions

- ◆ `int getsockname(int sock_fd, struct sockaddr *address, int *address_len)`
 - Returns the current address to which the socket is bound with using the buffer pointed to by address
- ◆ `int getpeername(int sock_fd, struct sockaddr *address, int *address_len)`
 - Get the name (address) of the peer connected to a socket; useful if a server has called a fork/exec combination and only the socket is known
 - On return, address points to struct sockaddr with fields filled in

The netstat command

- ◆ Use system program netstat to view the status of sockets
- ◆ E.g., netstat -n -l -p
 - Show in numerical form (-n)
 - the LISTEN-ing sockets (-l)
 - And the processes that own them (-p)
- ◆ Also...
 - -t / -u TCP/UDP
 - -a show sockets in all the states
- ◆ TCP states
 - LISTEN, ESTABLISHED, TIME_WAIT...

Parsing and Printing Addresses

Old:

- inet_ntoa** Convert struct `in_addr` to printable form 'a.b.c.d'
- inet_aton** Convert IP address string in '.' notation to 32bit network address

New (these work with IPv6 as well as IPv4):

- inet_ntop** Convert address from network format to printable presentation format
- inet_pton** Convert presentation format address to network format

Review

- ◆ A traditional way to write network servers:
 - the main server process (or thread) blocks on `accept()`, waiting for a connection.
 - once a connection comes in, the server forks new process (thread), the child process (thread) handles the connection
 - the main server process is able to accept new incoming requests.

Alternative server structure with `select()`

- ◆ A single process multiplexes all sockets via *select()* call
 - Adv: no need for separate child process/thread per request
- ◆ `Select()` blocks until an “event” occurs on a file descriptor (socket)
 - Data comes in off the net for a socket
 - File descriptor ready for writing
 - Describe events (file descriptors) of interest by filling a *fd_set* structure using macros

Alternative server structure with select () *

- ◆ Fill up a fd_set structure with the fds you want to know when data comes in on
- ◆ Fill up a fd_set structure with the fds you want to know when you can write on
- ◆ Call select(), block until event occurs
- ◆ When select returns, check if any fds was the reason process woke up and service relevant fd
- ◆ Repeat forever
- ◆ * See code samples at:

http://www.gnu.org/software/libc/manual/html_node/Server-Example.html#Server-Example

<http://www.lowtek.com/sockets/select.html>

```
#include "sockhelp.h"  
#include <ctype.h>  
#include <sys/time.h>  
#include <fcntl.h>
```

```
int sock;      /* The socket file descriptor for our "listening"  
               socket */  
int connectlist[5]; /* Array of connected sockets so we know  
                   who we are talking to */  
fd_set socks;    /* Socket file descriptors we want to wake  
                 up for, using select() */  
int highsock;   /* Highest #'d file desc, needed for select() */
```

```
void setnonblocking(sock)
```

```
int sock;  
{  
    int opts;  
  
    opts = fcntl(sock,F_GETFL);  
    if (opts < 0) {  
        perror("fcntl(F_GETFL)");  
        exit(EXIT_FAILURE);  
    }  
    opts = (opts | O_NONBLOCK);  
    if (fcntl(sock,F_SETFL,opts) < 0) {  
        perror("fcntl(F_SETFL)");  
        exit(EXIT_FAILURE);  
    }  
    return;  
}
```



```
void build_select_list() {
    int listnum;  /* Current item in connectlist for for loops */

    /* First put together fd_set for select(), which will
       consist of the sock variable in case a new connection
       is coming in, plus all the sockets we have already
       accepted. */
    /* FD_ZERO() clears out the fd_set called socks, so that
       it doesn't contain any file descriptors. */

    FD_ZERO(&socks);

    /* FD_SET() adds the file descriptor "sock" to the fd_set,
       so that select() will return if a connection comes in on
       that socket (which means you have to accept(), etc. */

    FD_SET(sock,&socks);

    /* Loops through all the possible connections and adds
       those sockets to the fd_set */

    for (listnum = 0; listnum < 5; listnum++) {
        if (connectlist[listnum] != 0) {
            FD_SET(connectlist[listnum],&socks);
            if (connectlist[listnum] > highsock)
                highsock = connectlist[listnum];
        }
    }
}
```

```

void handle_new_connection() {
    int listnum; /* Current item in connectlist for for loops */
    int connection; /* Socket file descriptor for incoming
                    connections */

    /* We have a new connection coming in! We'll
    try to find a spot for it in connectlist. */
    connection = accept(sock, NULL, NULL);
    if (connection < 0) {
        perror("accept");
        exit(EXIT_FAILURE);
    }
    setnonblocking(connection);
    for (listnum = 0; (listnum < 5) && (connection != -1); listnum++)
        if (connectlist[listnum] == 0) {
            printf("\nConnection accepted:  FD=%d; Slot=%d\n"
                    connection,listnum);
            connectlist[listnum] = connection;
            connection = -1;
        }
    if (connection != -1){
        /* No room left in the queue! */
        printf("\nNo room left for new client.\n");
        sock_puts(connection,"Sorry, this server is too busy. '
                    Try again later!\r\n");
        close(connection);
    }
}
}

```

```

void deal_with_data(
    int listnum  /* Current item in connectlist for for loops */
) {
    char buffer[80];  /* Buffer for socket reads */
    char *cur_char;  /* Used in processing buffer */

    if (sock_gets(connectlist[listnum],buffer,80) < 0) {
        /* Connection closed, close this end
           and free up entry in connectlist */
        printf("\nConnection lost: FD=%d; Slot=%d\n",
            connectlist[listnum],listnum);
        close(connectlist[listnum]);
        connectlist[listnum] = 0;
    } else {
        /* We got some data, so upper case it
           and send it back. */
        printf("\nReceived: %s; ",buffer);
        cur_char = buffer;
        while (cur_char[0] != 0) {
            cur_char[0] = toupper(cur_char[0]);
            cur_char++;
        }
        sock_puts(connectlist[listnum],buffer);
        sock_puts(connectlist[listnum],"");
        printf("responded: %s\n",buffer);
    }
}

```

```

void read_socks() {
    int listnum; /* Current item in connectlist for for loops */

    /* OK, now socks will be set with whatever socket(s)
       are ready for reading. Lets first check our
       "listening" socket, and then check the sockets
       in connectlist. */

    /* If a client is trying to connect() to our listening
       socket, select() will consider that as the socket
       being 'readable'. Thus, if the listening socket is
       part of the fd_set, we need to accept a new connection

    if (FD_ISSET(sock,&socks))
        handle_new_connection();
    /* Now check connectlist for available data */

    /* Run through our sockets and check to see if anything
       happened with them, if so 'service' them. */

    for (listnum = 0; listnum < 5; listnum++) {
        if (FD_ISSET(connectlist[listnum],&socks))
            deal_with_data(listnum);
    } /* for (all entries in queue) */
}

```

```

int main (argc, argv)
int argc;
char *argv[];
{
    char *ascport; /* ASCII version of the server port */
    int port; /* Port number after conversion from ascport */
    struct sockaddr_in server_address; /* bind info structure */
    int reuse_addr = 1; /* Used so we can re-bind to our port
                        while a previous connection is still
                        in TIME_WAIT state. */
    struct timeval timeout; /* Timeout for select */
    int readsocks; /* Number of sockets ready for reading */

    /* Make sure we got a port number as a parameter */
    if (argc < 2) {
        printf("Usage: %s PORT\r\n",argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Obtain a file descriptor for our "listening" socket */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    /* So that we can re-bind to it without TIME_WAIT
       problems */
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
               &reuse_addr, sizeof(reuse_addr));

```

```
/* Set socket to non-blocking with setnonblocking routine */  
setnonblocking(sock);  
  
/* Get the address information, and bind it to the socket */  
ascport = argv[1]; /* Read what the user gave us */  
port = atoi(ascport); /* Use function from sockhelp to  
convert to an int */  
memset((char *) &server_address, 0,  
sizeof(server_address));  
server_address.sin_family = AF_INET;  
server_address.sin_addr.s_addr = htonl(INADDR_ANY);  
server_address.sin_port = port;  
if (bind(sock, (struct sockaddr *) &server_address,  
sizeof(server_address)) < 0 ) {  
    perror("bind");  
    close(sock);  
    exit(EXIT_FAILURE);  
}  
  
/* Set up queue for incoming connections. */  
listen(sock,5);  
  
/* Since we start with only one socket, the listening socket,  
it is the highest socket so far. */  
highsock = sock;  
memset((char *) &connectlist, 0, sizeof(connectlist));
```

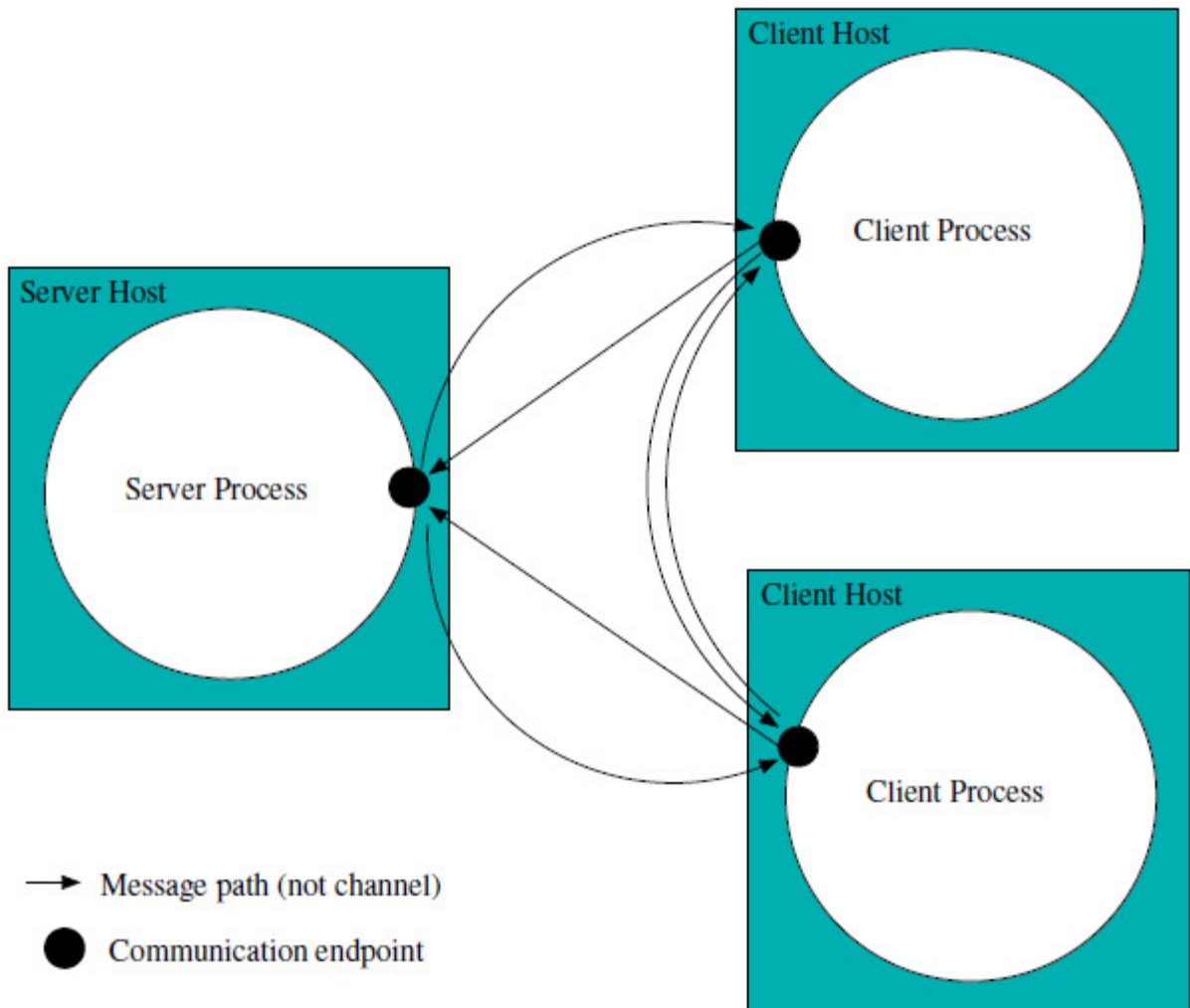
```
while (1) { /* Main server loop - forever */  
    build_select_list();  
    timeout.tv_sec = 1;  
    timeout.tv_usec = 0;  
  
    /* The first argument to select is the highest file  
       descriptor value plus 1. In most cases, you can  
       just pass FD_SETSIZE and you'll be fine. */  
  
    /* The second argument to select() is the address of  
       the fd_set that contains sockets we're waiting  
       to be readable (including the listening socket). */  
  
    /* The third parameter is an fd_set that you want to  
       know if you can write on -- this example doesn't  
       use it, so it passes 0, or NULL. The fourth parameter  
       is sockets you're waiting for out-of-band data for,  
       which usually, you're not. */  
  
    /* The last parameter to select() is a time-out of how  
       long select() should block. If you want to wait forever  
       until something happens on a socket, you'll probably  
       want to pass NULL. */  
  
    readsocks = select(highsock+1, &socks, (fd_set *) 0,  
                      (fd_set *) 0, &timeout);
```

/* select() returns the number of sockets that had things going on with them -- i.e. they're readable. */

/* Once select() returns, the original fd_set has been modified so it now reflects the state of why select() woke up. i.e. If file descriptor 4 was originally in the fd_set, and then it became readable, the fd_set contains file descriptor 4 in it. */

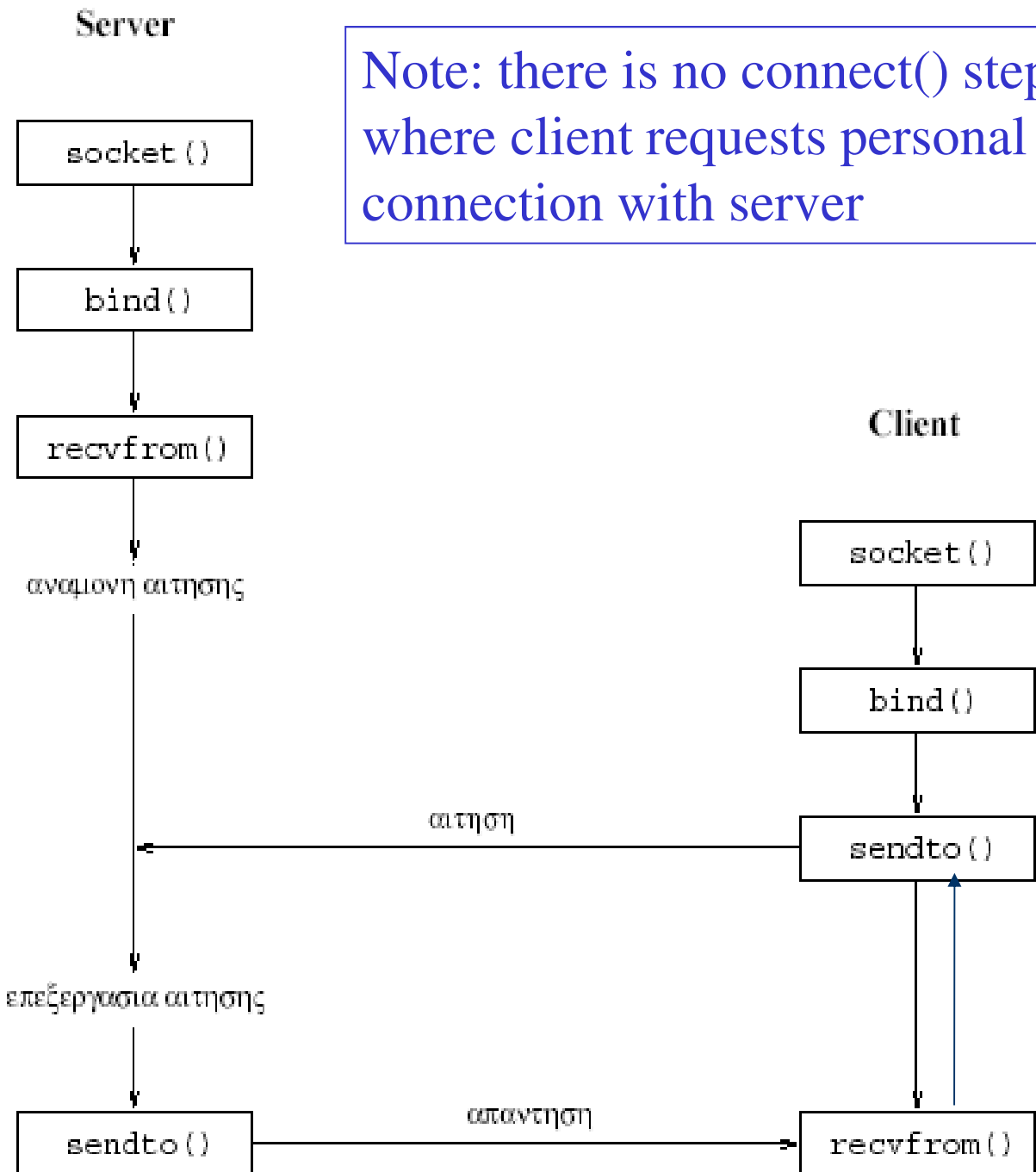
```
if (readsocks < 0) {  
    perror("select");  
    exit(EXIT_FAILURE);  
}  
if (readsocks == 0) {  
    /* Nothing ready to read, just show that  
       we're alive */  
    printf(".");  
    fflush(stdout);  
} else  
    read_socks();  
} /* while(1) */  
} /* main */
```


User Datagram Protocol (UDP)



- ◆ No Connections: Think postcards, not telephone.
- ◆ Datagrams (messages) exchanged.
- ◆ Datagrams either arrive (possibly out of order) or get lost!

UDP Communication



Note: there is no connect() step where client requests personal connection with server

sendto(), recvfrom()

- ◆ `ssize_t sendto (int sock , void *buff, int count, int flags , struct sockaddr *dest_addr, socklen_t dest_len);`
 - Send a message to a socket
 - Similar to `write()` & `send()` but designates destination
- ◆ `ssize_t recvfrom (int socket, void *buf, int count, int flags, struct sockaddr *address, socklen_t *address_len);`
 - **Receive a message** from a socket
 - Similar to `read()` & `recv()` but designates source address
 - address len is value-result and must be initialized to the size of the buffer pointed to by the address pointer
 - last two arguments can be NULL
- ◆ Usually `flags = 0` ; rarely used (ex. out of band data)

A simple echoing UDP server

Client on linux03:

```
mema@linux03> ./inet_dgr_client linux02 49024
```

Unix is cool

Unix is cool

Tax evasion SUCKS

Tax evasion SUCKS

Can I go back?

Can I go back?

```
mema@linux03>
```

Server on linux02:

```
mema@linux02> ./inet_dgr_server
```

Socket port : 49024

Received from linux03 : Unix is cool

Received from linux03 : Tax evasion SUCKS

Received from linux03 : Can I go back?

```

/* inet_dgr_server.c: Internet datagram sockets server */
#include <sys/types.h>                /* sockets */
#include <sys/socket.h>                /* sockets */
#include <netinet/in.h>                /* Internet sockets */
#include <netdb.h>                    /* gethostbyaddr */
#include <arpa/inet.h>                /* inet_ntoa */
#include <stdio.h>
#include <stdlib.h>
void perror_exit(char *message);
char *name_from_address(struct in_addr addr) {
    struct hostent *rem; int asize = sizeof(addr.s_addr);
    if ((rem = gethostbyaddr(&addr.s_addr, asize, AF_INET)))
        return rem->h_name; /* reverse lookup success */
    return inet_ntoa(addr); /* fallback to a.b.c.d form */
}
void main() {
    int n, sock; unsigned int serverlen, clientlen;
    char buf[256], *clientname;
    struct sockaddr_in server, client;
    struct sockaddr *serverptr = (struct sockaddr*) &server;
    struct sockaddr *clientptr = (struct sockaddr*) &client;
    /* Create datagram socket */
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        perror_exit("socket");
    /* Bind socket to address */
    server.sin_family = AF_INET; /* Internet domain */
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(0); /* Autoselect a port */
    serverlen = sizeof(server);

```

```

if (bind(sock, serverptr, serverlen) < 0)
    perror_exit("bind");
/* Discover selected port */
if (getsockname(sock, serverptr, &serverlen) < 0)
    perror_exit("getsockname");
printf("Socket port: %d\n", ntohs(server.sin_port));
while(1) { clientlen = sizeof(client);
    /* Receive message */
    if ((n = recvfrom(sock, buf, sizeof(buf), 0, clientptr,
        &clientlen)) < 0)
        perror("recvfrom");
    buf[sizeof(buf)-1]='\0'; /* force str termination */
    /* Try to discover client's name */
    clientname = name_from_address(client.sin_addr);
    printf("Received from %s: %s\n", clientname, buf);
    /* Send message */
    if (sendto(sock, buf, n, 0, clientptr, clientlen)<0)
        perror_exit("sendto");
    }}

```

```

void perror_exit(char *message)
{
    perror(message);
    exit(EXIT_FAILURE);
}

```

```
/* inet_dgr_client.c: Internet datagram sockets client */  
#include <sys/types.h> /* sockets */  
#include <sys/socket.h> /* sockets */  
#include <netinet/in.h> /* Internet sockets */  
#include <netdb.h> /* gethostbyname */  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>
```

```
void main(int argc, char *argv[]) {  
    int sock; char buf[256]; struct hostent *rem;  
    struct sockaddr_in server, client;  
    unsigned int serverlen = sizeof(server);  
    struct sockaddr *serverptr = (struct sockaddr *) &server;  
    struct sockaddr *clientptr = (struct sockaddr *) &client;  
    if (argc < 3) {  
        printf("Please give host name and port\n"); exit(1);}  
    /* Create socket */  
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {  
        perror("socket"); exit(1); }  
    /* Find server's IP address */  
    if ((rem = gethostbyname(argv[1])) == NULL) {  
        herror("gethostbyname"); exit(1); }  
    /* Setup server's IP address and port */  
    server.sin_family = AF_INET; /* Internet domain */  
    memcpy(&server.sin_addr, rem->h_addr, rem->h_length);  
    server.sin_port = htons(atoi(argv[2]));
```

```

/* Setup my address */
client.sin_family = AF_INET;      /* Internet domain */
client.sin_addr.s_addr=htonl(INADDR_ANY); /*Any address*/
client.sin_port = htons(0);      /* Autoselect port */
/* Bind my socket to my address*/
if (bind(sock, clientptr, sizeof(client)) < 0) {
    perror("bind"); exit(1); }
/* Read continuously messages from stdin */
while (fgets(buf, sizeof buf, stdin)) {
    buf[strlen(buf)-1] = '\0';      /* Remove '\n' */
    if (sendto(sock, buf, strlen(buf)+1, 0, serverptr,
                serverlen) < 0) {
        perror("sendto"); exit(1); } /* Send message */
    bzero(buf, sizeof buf);      /* Erase buffer */
    if (recvfrom(sock, buf, sizeof(buf), 0, NULL, NULL) < 0) {
        perror("recvfrom"); exit(1); } /* Receive message */
    printf("%s\n", buf); }
}

```


Looks good but....

- ◆ Everything looks good and runs ok, but there is a BUG!
- ◆ UDP is *unreliable*

```
/* rlsd.c - a remote ls server - with paranoia */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <string.h>
#include <ctype.h>
#define PORTNUM 15000 /* rlsd listens on this port */

void perror_exit(char *msg);
void sanitize(char *str);

int main(int argc, char *argv[]) {
    struct sockaddr_in myaddr; /* build our address here */
    int c, lsock, csock; /* listening and client sockets */
    FILE *sock_fp; /* stream for socket IO */
    FILE *pipe_fp; /* use popen to run ls */
    char dirname[BUFSIZ]; /* from client */
    char command[BUFSIZ]; /* for popen() */

    /** create a TCP a socket **/
    if ((lsock = socket( AF_INET, SOCK_STREAM, 0)) < 0)
        perror_exit( "socket" );
```

/ bind address to socket. */**

```
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);  
myaddr.sin_port = htons(PORTNUM); /*port to bind socket*/  
myaddr.sin_family = AF_INET; /* internet addr family */  
if(bind(lsock,(struct sockaddr*)&myaddr,sizeof(myaddr)))  
    perror_exit( "bind" );
```

/ listen for connections with Qsize=5 */**

```
if ( listen(lsock, 5) != 0 )  
    perror_exit( "listen" );  
while ( 1 ){ /* main loop: accept - read - write */  
    /* accept connection, ignore client address */  
    if ( (csock = accept(lsock, NULL, NULL)) < 0 )  
        perror_exit("accept");  
    /* open socket as buffered stream */  
    if ((sock_fp = fdopen(csock,"r+")) == NULL)  
        perror_exit("fdopen");  
    /* read dirname and build ls command line */  
    if (fgets(dirname, BUFSIZ, sock_fp) == NULL)  
        perror_exit("reading dirname");  
    sanitize(dirname);  
    snprintf(command, BUFSIZ, "ls %s", dirname);  
    /* Invoke ls through popen */  
    if ((pipe_fp = popen(command, "r")) == NULL )  
        perror_exit("popen");  
    /* transfer data from ls to socket */  
    while( (c = getc(pipe_fp)) != EOF )  
        putc(c, sock_fp);  
    pclose(pipe_fp);  
    fclose(sock_fp);  
}
```

(Fork+exec+pipe creation). Αντί για ανακατεύθυνση εντολής σε αρχείο, popen. Μπορώ να διαβάσω ή γράψω με popen. ΌΧΙ και τα 2 μαζί

```
return 0;
}
```

```
/* it would be very bad if someone passed us a dirname like  
* "; rm *" and we naively created a command "ls ; rm *".  
* So..we remove everything but slashes and alphanumerics.  
*/
```

```
void sanitize(char *str)
{
    char *src, *dest;
    for ( src = dest = str ; *src ; src++ )
        if ( *src == '/' || isalnum(*src) )
            *dest++ = *src;
    *dest = '\0';
}
```

```
/* Print error message and exit */
void perror_exit(char *message)
{
    perror(message);
    exit(EXIT_FAILURE);
}
```

```
/* rls.c - a client for a remote directory listing service
```

```
*      usage: rls hostname directory */
```

```
#include <stdio.h>
```

```
#include <stdlib.h>          /* exit */
```

```
#include <string.h>        /* strlen */
```

```
#include <unistd.h>        /* STDOUT_FILENO */
```

```
#include <sys/types.h>     /* sockets */
```

```
#include <sys/socket.h>    /* sockets */
```

```
#include <netinet/in.h>    /* internet sockets */
```

```
#include <netdb.h>         /* gethostbyname */
```

```
#define PORTNUM 15000
```

```
#define BUFFSIZE 256
```

```
void perror_exit(char *msg);
```

```
/* Write() repeatedly until 'size' bytes are written */
```

```
int write_all(int fd, void *buff, size_t size) {
```

```
    int sent, n;
```

```
    for(sent = 0; sent < size; sent+=n) {
```

```
        if ((n = write(fd, buff+sent, size-sent)) == -1)
```

```
            return -1; /* error */
```

```
    }
```

```
    return sent;
```

```
}
```

```
int main(int argc, char *argv[]) {
```

```
    struct sockaddr_in servadd; /* The address of server */
```

```
    struct hostent *hp;        /* to resolve server ip */
```

```
    int sock, n_read;         /* socket and message length */
```

```
    char buffer[BUFFSIZE];    /* to receive message */
```

```

if ( argc != 3 ) {
    puts("Usage: rls <hostname> <directory>");exit(1);}
/* Step 1: Get a socket */
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1 )
    perror_exit( "socket" );
/* Step 2: lookup server's address and connect there */
if ((hp = gethostbyname(argv[1])) == NULL) {
    perror("gethostbyname"); exit(1);}
memcpy(&servadd.sin_addr, hp->h_addr, hp->h_length);
servadd.sin_port = htons(PORTNUM); /* set port number */
servadd.sin_family = AF_INET ; /* set socket type */
if (connect(sock, (struct sockaddr*) &servadd,
            sizeof(servadd)) !=0)
    perror_exit( "connect" );
/* Step 3: send directory name + newline */
if ( write_all(sock, argv[2], strlen(argv[2])) == -1)
    perror_exit("write");
if ( write_all(sock, "\n", 1) == -1 )
    perror_exit("write");
/* Step 4: read back results and send them to stdout */
while( (n_read = read(sock, buffer, BUFSIZE)) > 0 )
    if (write_all(STDOUT_FILENO, buffer, n_read)<n_read)
        perror_exit("fwrite");
close(sock);
return 0;
}

```

Server on linux01

k24-syspro@linux01> ./rld

ls: cannot open directory /home/users/mema: Permission denied

Client on linux02

mema@linux02> ./rld linux01.di.uoa.gr /home/users/k24-syspro

project1

project2

project3

students.txt

mema@linux02> ./rld linux01.di.uoa.gr /home/users/mema/

mema@linux02>

One more example

The **ROCK PAPER SCISSORS** game

- One referee process.
- Two players: a local process, a remote process
- Referee talks to the local process through pipes
- Referee talks to the remote process through sockets

Referee (prsref) program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>          /* For wait */
#include <sys/types.h>        /* For sockets */
#include <sys/socket.h>       /* For sockets */
#include <netinet/in.h>       /* For Internet sockets */
#include <netdb.h>            /* For gethostbyname */
```

```
#define READ  0
#define WRITE 1
```

```
int read_data (int fd, char *buffer);
int write_data (int fd, char* message);
void prs (int *score1, int *score2, int len1, int len2);
```

```
int main(int argc, char *argv[])
{
    int n, port, sock, newsock;
    int i, pid, fd1[2], fd2[2], option, status;
    int score1=0, score2=0;      /* Score variables */
    char buf[60], buf2[60], buf3[60]; /* Buffers */
    /* prs options */
    char *message[] = { "ROCK", "PAPER", "SCISSORS" };
    unsigned int serverlen, clientlen; /* Server- client variables */
    struct sockaddr_in server, client;
    struct sockaddr *serverptr, *clientptr;
    struct hostent *rem;
```

```

if ( argc < 3 ){
    /* At least 2 arguments */
    fprintf(stderr, "usage: %s <n> <port>\n", argv[0]);
    exit(0);
}



---


n = atoi(argv[1]); /* Number of games */
port = atoi(argv[2]); /* Port */
/* Create socket */
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1){

    perror("socket");
    exit(-1);
}
server.sin_family = AF_INET; /* Internet domain */
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons(port); /* The given port */
serverptr = (struct sockaddr *) &server;
serverlen = sizeof server;
if (bind(sock, serverptr, serverlen) < 0){
    perror("bind"); exit(-1);
}
if (listen(sock, 5) < 0){
    perror("listen");exit(-1);
}

printf("I am the referee with PID %d waiting for game
request at port %d\n", (int) getpid(), port);

if (pipe (fd1) == -1){ /* First pipe: parent -> child */
    perror("pipe");exit(-1);
}

```

```

}
if (pipe(fd2) == -1){ /* Second pipe: child -> parent */
    perror("pipe");exit(-1);
}



---


if ((pid = fork()) == -1) /* Create child for player 1 */
{
    perror("fork");exit(-1);
}

if ( !pid ){ /* Child process */
    close(fd1[WRITE]);close(fd2[READ]); /*Close unused*/
    srand(getppid());
    printf("I am player 1 with PID %d\n", (int) getpid());
    for(;;) /* While read "READY" */
    {
        /* Read "READY" or "STOP" */
        read_data(fd1[READ], buf);
        option = rand() %3;
        if ( strcmp("STOP", buf)){ /* If != "STOP" */
            /* Send random option */
            write_data(fd2[WRITE], message[option]);
            /* Read result of this game */
            read_data(fd1[READ], buf);
            printf("%s", buf); /* Print result */
        }else
            break;
    }
    read_data(fd1[READ], buf); /* Read final result */

```

```

printf("%s", buf);  /* Print final result */
close(fd1[READ]); close(fd2[WRITE]);
}
else{  /* Parent process */
  clientptr = (struct sockaddr *) &client;
  clientlen = sizeof client;
  close(fd1[READ]); close(fd2[WRITE]);
  printf("Player 1 is child of the referee\n");
  if ((newsock = accept(sock, clientptr, &clientlen)) < 0){
    perror("accept"); exit(-1);
  }
  if ((rem = gethostbyaddr((char *) &client.sin_addr.s_addr,
    sizeof client.sin_addr.s_addr, client.sin_family))
    == NULL) {
    perror("gethostbyaddr");exit(-1);
  }

  printf("Player 2 connected %s\n",rem->h_name);
  write_data (newsock, "2"); /* Send player's ID (2) */
  for(i = 1; i <= n; i++){
    write_data(fd1[WRITE], "READY");
    write_data(newsock, "READY");
    read_data(fd2[READ], buf);
    read_data(newsock, buf2);
    /* Create result string */
    sprintf(buf3, "Player 1:%10s\tPlayer 2:%10s\n",
      buf, buf2);
    write_data(fd1[WRITE], buf3);
    write_data(newsock, buf3);
    prs(&score1,&score2,strlen(buf),strlen(buf2));

```

```
}
```

```
/* Calculate final results for each player */
```

```
if ( score1 == score2 ){
```

```
    printf(buf, "Score = %d - %d (draw)\n",  
        score1, score2);
```

```
    printf(buf2, "Score = %d - %d (draw)\n",  
        score1, score2);
```

```
}else if (score1 > score2 ){
```

```
    printf(buf, "Score = %d - %d (you won)\n",  
        score1, score2);
```

```
    printf(buf2, "Score = %d - %d (player 1 won)\n"  
        score1, score2);
```

```
}else{
```

```
    printf(buf, "Score = %d - %d (player 2 won)\n",  
        score1, score2);
```

```
    printf(buf2, "Score = %d - %d (you won)\n",  
        score1, score2);
```

```
}
```

```
write_data(fd1[WRITE], "STOP");
```

```
write_data(fd1[WRITE], buf);
```

```
close(fd1[WRITE]); close(fd2[READ]);
```

```
wait(&status); /* Wait child */
```

```
write_data(newsock, "STOP");
```

```
write_data(newsock, buf2);
```

```
close(newsock); /* Close socket */
```

```
}
```

```
return 0;
```

```
}
```

```

int read_data(int fd, char *buffer){ /* Read formatted data*/
    char temp;int i = 0, length = 0;
    if ( read(fd, &temp, 1 ) < 0 ) /* Get length of string */
        exit (-3);
    length = temp;
    while ( i < length ) /* Read $length chars */
        if ( i < ( i+= read(fd, &buffer[i], length - i)))
            exit (-3);
    return i; /* Return size of string */
}

```

/*Write formatted data*/

```

int write_data( int fd, char* message ){
    char temp; int length = 0;
    length = strlen(message) + 1; /* Find length of string */
    temp = length;
    if( write(fd, &temp, 1) < 0 ) /* Send length first */
        exit(-2);
    if( write(fd, message, length) < 0 )/* Send string */
        exit(-2);
    return length; /* Return size of string */
}

```

```

/* void prs(int *score1, int *score2, int len1, int len2):
* Each option (PAPER, ROCK, SCISSORS) has a number of letters (5, 4, 8).
* PAPER wins ROCK, ROCK wins SCISSORS and SCISSORS win PAPER.
* This means, for the 1st player to be the winner the difference in the
* number of letters must be equal to 3 (SCISSORS-PAPER)
* or 1 (PAPER-ROCK)
* or -4 (ROCK-SCISSORS). If not, then the 2nd player wins!
* (If we have a zero, then we call it a draw and nobody get points)
*/
void prs(int *score1, int *score2, int len1, int len2)
{
    /* len1 = buf1 length, len2 = buf2 length */
    int result = len1 - len2;
    if (result == 3 || result == 1 || result == -4) /* 1st player win
        (*score1)++;
    else if (result) /* 2nd player wins */
        (*score2)++;
    return;
}

```

Remote player (prs)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>      /* For bcopy */
#include <unistd.h>
#include <sys/wait.h>     /* For wait */
#include <sys/types.h>    /* For sockets */
#include <sys/socket.h>   /* For sockets */
#include <netinet/in.h>   /* For Internet sockets */
#include <netdb.h>        /* For gethostbyname */

int read_data(int fd, char *buffer);
int write_data(int fd, char* message);

int main (int argc, char *argv[])
{
    int i, port, sock, option;
    char opt[3], buf[60], *message[] =
        { "PAPER", "ROCK", "SCISSORS" };
    unsigned int serverlen;
    struct sockaddr_in server;
    struct sockaddr *serverptr;
    struct hostent *rem;
    if (argc < 3){ /* At least 2 arguments */
        fprintf(stderr, "usage: %s <domain> <port>\n",
            argv[0]);
        exit(-1);
    }
}
```



```

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0){
    perror("socket");
    exit(-1);
}
/* Find server address */
if ((rem = gethostbyname(argv[1])) == NULL){
    perror("gethostbyname");
    exit(-1);
}
port = atoi(argv[2]);
server.sin_family = AF_INET;
bcopy((char *) rem -> h_addr, (char *) &server.sin_addr,
      rem -> h_length);
server.sin_port = htons(port);
serverptr = (struct sockaddr *) &server;
serverlen = sizeof server;
if (connect(sock, serverptr, serverlen) < 0){
    perror("connect");exit(-1);
}

read_data(sock, buf); /* Read player's ID (1 or 2) */
printf("I am player %d with PID %d\n", buf[0]-'0',
      (int) getpid());
for ( i = 1; ; i++ ){ /* While read "READY" */
    read_data(sock, buf);/* Read "READY" or "STOP"*/
    if ( strcmp("STOP", buf) ){ /* If != "STOP" */
        printf("Give round %d play: ", i);
        scanf("%s", opt);
    }
}

```

```

switch (*opt){    /* First letter of opt */
/* Note: The other 2 are \n and \0 */
    case 'p':option = 0; break;
    case 'r':option = 1; break;
    case 's':option = 2; break;
    default: fprintf(stderr, "Wrong
                    option %c\n", *opt);
                option = ((int)*opt)%3; break;
    }
    write_data (sock, message[option]);
    read_data (sock, buf);
    printf ("%s", buf);
}
else break;
}
read_data (sock, buf);    /* Read final score */
printf("%s", buf);
close(sock);
return 0;
}

```

/* Read formatted data */

```

int read_data (int fd, char *buffer){
    char temp; int i = 0, length = 0;
    if ( read ( fd, &temp, 1 ) < 0 ) /* Get length of string */
        exit (-3);
    length = temp;
    while ( i < length )    /* Read $length chars */
        if ( i < ( i+= read (fd, &buffer[i], length - i) ) )
            exit (-3);
}

```

```
    return i;    /* Return size of string */
}
```

```
/* Write formatted data */
```

```
int write_data ( int fd, char* message ){
    char temp; int length = 0;
    length = strlen(message) + 1; /* Find length of string */
    temp = length;
    if ( write (fd, &temp, 1) < 0 ) /* Send length first */
        exit (-2);
    if ( write (fd, message, length) < 0 ) /* Send string */
        exit (-2);
    return length; /* Return size of string */
}
```

Server

mema@browser> ./prsref 3 2323

I am the referee with PID 25499 ...

waiting for game request at port 2323

I am player 1 with PID 25500

Player 1 is child of the referee

Player 2 connected localhost

Player 1: PAPER Player 2: PAPER

Player 1: ROCK Player 2: SCISSORS

Player 1: SCISSORS Player 2: SCISSORS

Score = 1 - 0 (you won)

mema@browser>

Client

mema@browser> ./prs localhost 2323

I am player 2 with PID 25519

Give round 1 play: p

Player 1: PAPER Player 2: PAPER

Give round 2 play: s

Player 1: ROCK Player 2: SCISSORS

Give round 3 play: s

Player 1: SCISSORS Player 2: SCISSORS

Score = 1 - 0 (player 1 won)

mema@browser>