Internetworking with Sockets
Cross-host Interprocess Communication (IPC)

- Typically client-server model over network
- Server - Provides a service
- Server - Waits for clients to connect
- Clients - Connect to utilize the service
- Clients - Possibly more than one at a time
The Internet Protocol

- Each device in a network is assigned an IP address
- IPv4 32 bit, IPv6 128 bit
- 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
Common Service Use Cases

Browse the World Wide Web

- Each device has a static IP
- DNS used to translate www.google.com to 74.125.43.103
- Contact service at 74.125.43.103 and port 80 (http)
Common Service Use Cases

Your home network.

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

- TCP uses acknowledgments
- Non-acknowledged messages are retransmitted
- Messages re-ordered by the receiver’s OS network stack
- Application sees a properly ordered data stream

![Diagram of TCP connection]

- Passive communication endpoint
- Communication endpoint
TCP - multiple clients

Server Host

Client Host

Client Process

Server Process

- Passive communication endpoint
- Communication endpoint
Sockets

- A *socket* is a communication endpoint.
- Processes refer to a socket using an integer descriptor.
- Communication domain:
  - Internet domain (over internet)
  - Unix domain (same host)
- Communication type:
  - Stream (usually TCP)
  - Datagram (usually UDP)
<table>
<thead>
<tr>
<th>Feature</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection Required</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Reliability</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Message Boundaries</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>In-Order Data Delivery</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Socket Type</td>
<td>SOCK_STREAM</td>
<td>SOCK_DGRAM</td>
</tr>
<tr>
<td>Socket Domain</td>
<td>Internet</td>
<td>Internet</td>
</tr>
<tr>
<td>Latency</td>
<td>higher</td>
<td>lower</td>
</tr>
<tr>
<td>Flow Control</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
Serial Server (TCP)

Create listening socket $a$

```
loop
  Wait for client request on $a$
  Open two-way channel $b$ with client
  while request received through $b$ do
    Process request
    Send response through $b$
  end while
  Close file descriptor of $b$
end loop
```

Drawbacks:

- Serves only one client at a time
- 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
- Inefficient, too many clients → too many processes
1 process per client model

Parent process

Create listening socket \( a \)

\textbf{loop}

Wait for client request on \( a \)
Create two-way channel \( b \) with client
Fork a child to handle the client
Close file descriptor of \( b \)

\textbf{end loop}

Child process

Close listening socket \( a \)
Serve client requests through \( b \)
Close private channel \( b \)
Exit
Parent process: why close file descriptor \( b \)?

- Parent doesn’t need this file descriptor
- Risk of running out of file descriptors otherwise
- 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

- Multiple threads handle multiple clients concurrently
- Drawback: Requires synchronization for access to shared resources
Dealing with byte order

- Byte order poses a problem for the communication among different architectures.
- Convention: ip addresses, port numbers etc. in *Network Byte Order*
- Convert long/short integers between *Host* and *Network* byte order

```c
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```
From Domain Names to Addresses and back

- An *address* is needed for network communication
- We often have to *resolve* the address from a domain name.
  ex. spiderman.di.uoa.gr ↔ 195.134.66.107

```c
struct hostent {
    char  *h_name;     /* official name of host */
    char  **h_aliases; /* aliases (alt. names) */
    int   h_addrtype;  /* usually AF_INET */
    int   h_length;    /* bytelength of address */
    char  **h_addr_list; /* list of addresses */
};

struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);
```

- Beware, both use static storage for struct hostent. (problem?)
- For error reporting use herror & hstrerror
Our goal

Create the communication endpoint. Use it as a file descriptor.

**Server Process**
- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `wait for connection`
- `read()`
- `processing`
- `write()`

**Client Process**
- `socket()`
- `connect()`
- `write()`
- `request`
- `response`
- `read()`
Creating sockets

- `socket` creates an endpoint for communication
- returns a descriptor or -1 on error

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

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

- **domain** communication domain (usual. PF_INET)
- **type** communication semantics (usual. SOCK_STREAM, SOCK_DGRAM)
- **protocol** Use 0 as typically only one protocol is available

```c
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    perror("Socket creation failed!");
```
Binding sockets to addresses

- *bind* requests for an address to be assigned to a socket
- You must bind a SOCK_STREAM socket to a local address before receiving connections

```c
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

Internet domain (AF_INET):
- We pass a `sockaddr_in` struct as the *address*

Interesting fields:

- `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

```c
#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**

```c
int listen(int socket, int backlog);
```
- Listen for connections on a socket
- At most `backlog` connections will be queued waiting to be accepted

```c
int accept(int socket, struct sockaddr *address, socklen_t *address_len);
```
- Accept a connection on a socket
- Blocks until a client connects or interrupted by signal
- Returns new socket descriptor used 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

```c
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).
TCP connection

Server Process

socket()

bind()

listen()

accept()

wait for connection

read()

processing

write()

Client Process

socket()

connect()

request for connection establishment

write()

request

response

read()
Tips and warnings

- In Solaris compile with “-lsocket -lnsl”
- If a process attempts to write through a socket that has been closed by the other peer, a SIGPIPE signal is received.
- SIGPIPE is by default fatal, install a signal handler to override this.
- Use `netstat` to view the status of sockets.

```
ad@linux03:~$ netstat -ant
```

- 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
TCP server that receives a string and replies with the string capitalized.

```c
/* 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 newsock);
void perror_exit(char *message);
void sigchld_handler (int sig);

void main(int argc, char *argv[]) {
    int port, sock, newsock;
    struct sockaddr_in server, client;
    socklen_t clientlen = sizeof(client);
    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) {
    /* accept connection */
    if ((newsock = accept(sock, clientptr, &clientlen)) < 0) perror_exit("accept");
    /* Find client’s address */
    printf("Accepted connection\n");
    switch (fork()) {
        /* Create child for serving client */
        case -1: /* Error */
            perror("fork"); break;
        case 0: /* Child process */
            close(sock); child_server(newsock);
            exit(0);
    }
    close(newsock); /* parent closes socket to client */
}
```c
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);
}
```
TCP client example. (definitions)

/* 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);
    }
TCP client example. (connection)

```c
/* 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);
```
TCP client example. (transfer loop)

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);
}
Execution

Server on linux02:

```
ad@linux02:~> ./server 9002
Listening for connections to port 9002
Accepted connection from linux03.di.uoa.gr
Hello world
EnD
Closing connection.
```

Client on linux03:

```
ad@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
ad@linux03:~>
```
More useful functions

**shutdown**  shut down part of a full-duplex connection

```c
int shutdown(int socket, int how);
```

Can be used to tell server that we have sent the whole request.

**getsockname**  get the current address of a socket

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

**getpeername**  get the name (address) of the peer connected to socket. (inverse of getsockname)

```c
int getpeername(int socket, struct sockaddr * address, socklen_t *address_len);
```
Parsing and Printing Addresses

- **inet_ntoa**: Convert struct in_addr to printable form ‘a.b.c.d’
- **inet_addr**: Convert IP address string in ‘.’ notation to 32bit network address
- **inet_ntop**: Convert address from network format to printable presentation format
- **inet_ntop**: Convert presentation format address to network format

**Bonus**: `inet_ntop` and `inet_ntop` also work with IPv6!
Internet User Datagram Protocol (UDP)

- no connections. Think postcards, not telephone.
- *Datagrams* (messages) exchanged.
- Datagrams get lost or arrive out of order
UDP communication

Server
- socket()
- bind()
- recvfrom()
- wait for message
- sendto()
- processing

Client
- socket()
- bind()
- sendto()
- request
- response
- recvfrom()
sendto, recvfrom

```c
ssize_t sendto(int sock, void *buff, size_t length,
               int flags, struct sockaddr *dest_addr, socklen_t dest_len);
```

- Send a message from a socket
- Similar to write() & send() (which we haven’t seen) but allows to specify destination

```c
ssize_t recvfrom(int socket, void *buff, size_t length,
                 int flags, struct sockaddr *addr, socklen_t *address_len);
```

- Receive a message from a socket
- Similar to read() & recv() (which we haven’t seen) but allows to get the 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 (e.g., Out Of Band data)
A simple echoing UDP server

Client on linux03 (needs to be given the server’s port#):

```
ad@linux03:~> fortune | ./inet_dgr_client linux02 59579
Hlade’s Law:
    If you have a difficult task, give it to a lazy person --
    they will find an easier way to do it.
ad@linux03:~>
```

Server on linux02:

```
ad@linux02:~> ./inet_dgr_server
Socket port: 59579
Received from linux03: Hlade’s Law:
Received from linux03: If you have a difficult task, give
it to a lazy person --
Received from linux03: they will find an easier way to do
it.
```
/* 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 <netinet/in.h>  /* Internet sockets */
#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");
}}
/* 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) {
        perror("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(sockfd, (struct sockaddr *)&client, 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(sockfd, buf, strlen(buf)+1, 0, (struct sockaddr *)&server, serverlen) < 0) {
        perror("sendto"); exit(1); } /* Send message */
    bzero(buf, sizeof(buf)); /* Erase buffer */
    if (recvfrom(sockfd, buf, sizeof(buf), 0, NULL, NULL) < 0) {
        perror("recvfrom"); exit(1); } /* Receive message */
    printf("%s\n", buf);
}
}
Everything looks good and runs ok BUT there is a **BUG**!

Remember that UDP is *unreliable*
rlsd: a remote ls server - with paranoia

Server on linux02:

ad@linux02:~> ./rlsd

Client on linux03:

ad@linux03:~> ./rls linux02.di.uoa.gr /usr/share/dict
README
connectives
propernames
web2
web2a
words
ad@linux03:~>
rlsd.c  remote ls server with paranoia (TCP)

Demonstrates interesting calls

fdopen  allows buffered I/O by opening socket as file stream
popen  pipe+fork+execute command together

/* 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 <netinet/tcp.h>
#include <ctype.h>
#define PORTNUM 15000  /* rlsd listens on this port */

void perror_exit(char *msg);
void sanitize(char *str);
```c
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);
}
return 0;
/ * it would be very bad if someone passed us an 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 sends a directory name to rlsd and reads back a directory listing (TCP)

**write_all** guarantees to send all the bytes requested, provided no error occurs, by repeatedly calling write()

```c
#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) {
        herror("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, BUFFSIZE)) > 0)
    if (write_all(STDOUT_FILENO, buffer, n_read)<n_read)
        perror_exit("fwrite");
close(sock);
return 0;
The ROCK PAPER SCISSORS game

- One referee process.
- Two players: a local process (playing random), a remote process (accepting input)
- Referee talks to the local process through pipes
- Referee talks to the remote process through sockets
Server

```c
#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 */
    char *message[] = { "ROCK", "PAPER", "SCISSORS" }; /* prs options */

    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 */
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1){
    /* Create socket */
    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_data(fd1[READ], buf); /* Read "READY" or "STOP" */
        option = rand()%3;
        if ( strcmp("STOP", buf)){ /* If != "STOP" */
            write_data(fd2[WRITE], message[option]); /* Send random option */
            read_data(fd1[READ], buf); /* Read result of this game */
            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 ){
    sprintf(buf, "Score = %d - %d (draw)\n", score1, score2);
    sprintf(buf2, "Score = %d - %d (draw)\n", score1, score2);
}else if (score1 > score2 ){
    sprintf(buf, "Score = %d - %d (you won)\n", score1, score2);
    sprintf(buf2, "Score = %d - %d (player 1 won)\n", score1, score2);
}else{
    sprintf(buf, "Score = %d - %d (player 2 won)\n", score1, score2);
    sprintf(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 */
```c
int read_data (int fd, char *buffer){ /* Read formated 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 */
}

int write_data ( int fd, char* message ){ /* Write formated data */
    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 */
}
```
```c
void prs(int *score1, int *score2, int len1, int len2) /*
Calculate score */
{
    int result = len1 - len2; /* len1 = buf1 length, len2 = buf2 length */
    if (result == 3 || result == 1 || result == -4) /* 1st player wins */
        (*score1)++;
    else if (result) /* 2nd player wins */
        (*score2)++;
    return;
}
```
#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>
", argv[0]);
        exit(-1);
    }
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        perror("socket");
        exit(-1);
    }
    if ((rem = gethostbyname(argv[1])) == NULL){ /* Find server address */
        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;
Server

```
jackal@jackal-laptop:~/Set006/src$ ./prsref 3 2323
I am the referee with PID 4587 waiting for game request at port 2323
I am player 1 with PID 4588
Player 1 is child of the referee
Player 2 connected localhost
Player 1: PAPER    Player 2: PAPER
Player 1: SCISSORS Player 2: SCISSORS
Player 1: ROCK     Player 2: SCISSORS
Score = 1 - 0 (you won)
```

Client

```
jackal@jackal-laptop:~/Set006/src$ ./prs localhost 2323
I am player 2 with PID 4615
Give round 1 play: p
Player 1: PAPER    Player 2: PAPER
Give round 2 play: s
Player 1: SCISSORS Player 2: SCISSORS
Give round 3 play: s
Player 1: ROCK     Player 2: SCISSORS
Score = 1 - 0 (player 1 won)
```