Linux System Programming Part 7 - Sockets

IBA Bulgaria
2018

Sockets in Linux

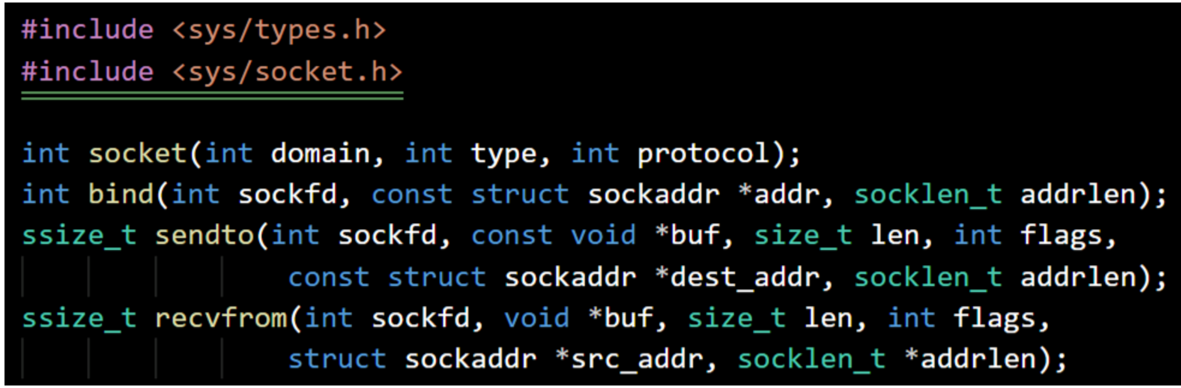
Most interprocess communication uses the ***client-server*** model. One of the two processes, the ***client***, connects to the other process, the ***server***, typically to make a request for information. Each end of such communication can be represented in IPC with a **socket**. The client and server processes each establish their own **socket**.

- A **Unix domain socket** is a data communications endpoint for exchanging data between processes executing on the same host operating system.
- A **network socket** is an internal endpoint for sending or receiving data within a node on a computer network.
- Sockets support transmission of a reliable **stream** of bytes, as well as ordered and reliable transmission of **datagrams**. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once.

Working with Sockets in C

- **socket**() - creates an endpoint for communication.
- **bind**() - binds a name to a socket.
- **recv**(), **recvfrom**(), **recvmsg**() - receives a message from a socket.
- **send**(), **sendto**(), **sendmsg**() - sends a message on a socket.
- **socketpair**() - creates a pair of connected sockets.
- **listen**() - listens for connections on a socket.
- **accept**() - accepts a connection on a socket.
- **connect**() - initiates a connection on a socket.
- **htons**() - converts unsigned short integer to network byte order.
- **gethostbyname**() - returns a structure of type hostent for the given host name.

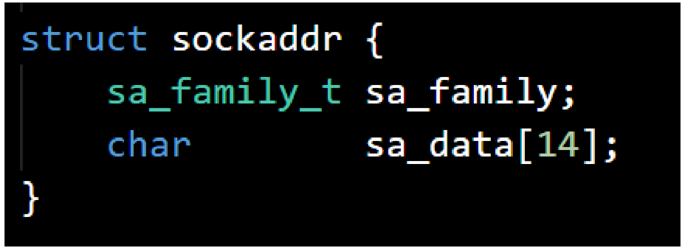
Unix Sockets usage



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

int socket(int domain, int type, int protocol);
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                struct sockaddr *src_addr, socklen_t *addrlen);
```

- **socket**() creates an endpoint for communication and returns a file descriptor that refers to that endpoint.
  - **domain** specifies the protocol family which will be used (**AF_UNIX**, **AF_LOCAL**, **AF_INET**, etc.).
  - **type** specifies the communication semantics (**SOCK_STREAM**, **SOCK_DGRAM**, etc.).
- When a socket is created with **socket**(), it exists in a name space (address family) but has no address assigned to it.

- **bind**() assigns the address specified by **addr** to the socket referred to by the file descriptor **sockfd** (returned from **socket**()).
- **sockaddr** is the following structure:



```c
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
}
```

fsserver.c

File socket server

Open Unix datagram **socket**, exit on error

Bind **socket** with ‘**socket.soc**’, exit on error

Get a message from **socket**, exit on error

Print the message

Close the socket

Remove the socket file

```c
#define SOCK_NAME "socket.soc"
...
struct sockaddr srvr_name, rcvr_name;
...
socklen_t namelen;
sock = socket(AF_UNIX, SOCK_DGRAM, 0);
if (sock < 0){ // ERROR
...
srvr_name.sa_family = AF_UNIX;
strcpy(srvr_name.sa_data, SOCK_NAME);
if (bind(sock, &srvr_name, strlen(srvr_name.sa_data) +
    sizeof(srvr_name.sa_family)) < 0){ //ERROR
...
namelen = sizeof(srvr_name);
bytes = recvfrom(sock, buf, sizeof(buf),  0,
                    &rcvr_name, &namelen);
if (bytes < 0) { //ERROR
...
buf[bytes] = 0;
rcvr_name.sa_data[namelen] = 0;
printf("Client sent: %s\n", buf);
close(sock);
unlink(SOCK_NAME);
```

fsclient.c

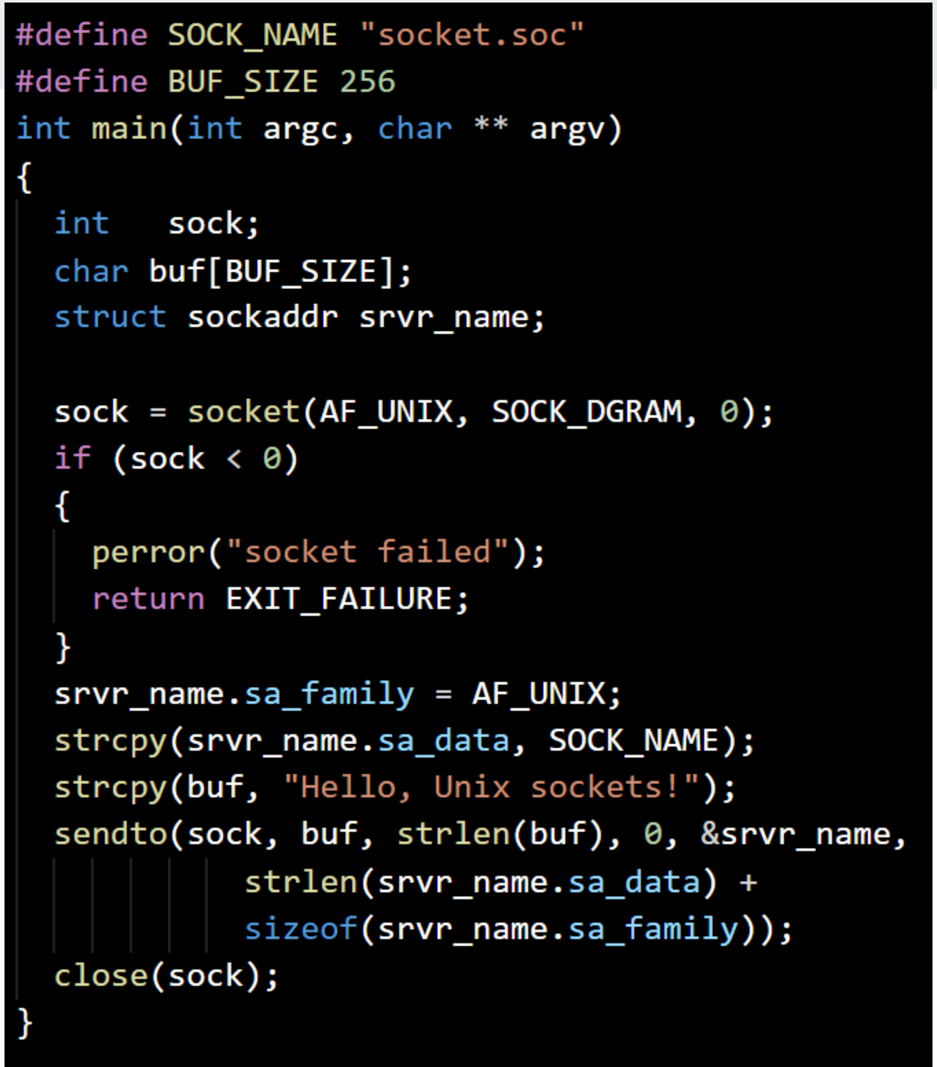
File socket client

Open Unix datagram **socket**, exit on error

Setup the server name structure for Unix socket and name ‘**socket.soc**’

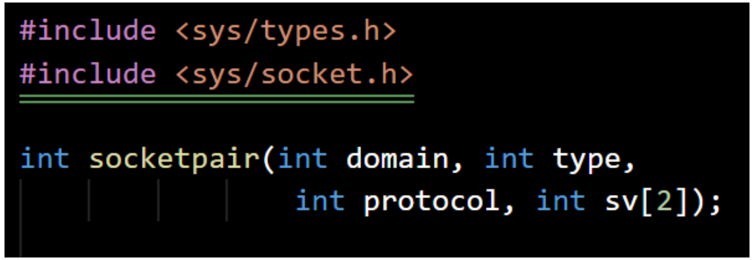
Send a message to the server

Close the socket



```c
#define SOCK_NAME "socket.soc"
#define BUF_SIZE 256
int main(int argc, char ** argv)
{
  int   sock;
  char buf[BUF_SIZE];
  struct sockaddr srvr_name;

  sock = socket(AF_UNIX, SOCK_DGRAM, 0);
  if (sock < 0)
  {
    perror("socket failed");
    return EXIT_FAILURE;
  }
  srvr_name.sa_family = AF_UNIX;
  strcpy(srvr_name.sa_data, SOCK_NAME);
  strcpy(buf, "Hello, Unix sockets!");
  sendto(sock, buf, strlen(buf), 0, &srvr_name,
            strlen(srvr_name.sa_data) +
            sizeof(srvr_name.sa_family));
  close(sock);
}
```

Sockets pair



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

int socketpair(int domain, int type,
               int protocol, int sv[2]);
```

The socketpair() call creates an unnamed pair of connected sockets in the specified **domain**, of the specified **type**, and using the optionally specified **protocol**.

The file descriptors used in referencing the new sockets are returned in **sv[0]** and **sv[1]**. The two sockets are indistinguishable.

On success, ***zero*** is returned. On error, ***-1*** is returned, and ***errno*** is set appropriately.

sockpair.c

Unix Sockets Pair

Open a sockets pair, exit on error

Fork the process

<u>The parent</u>: close the “second” socket

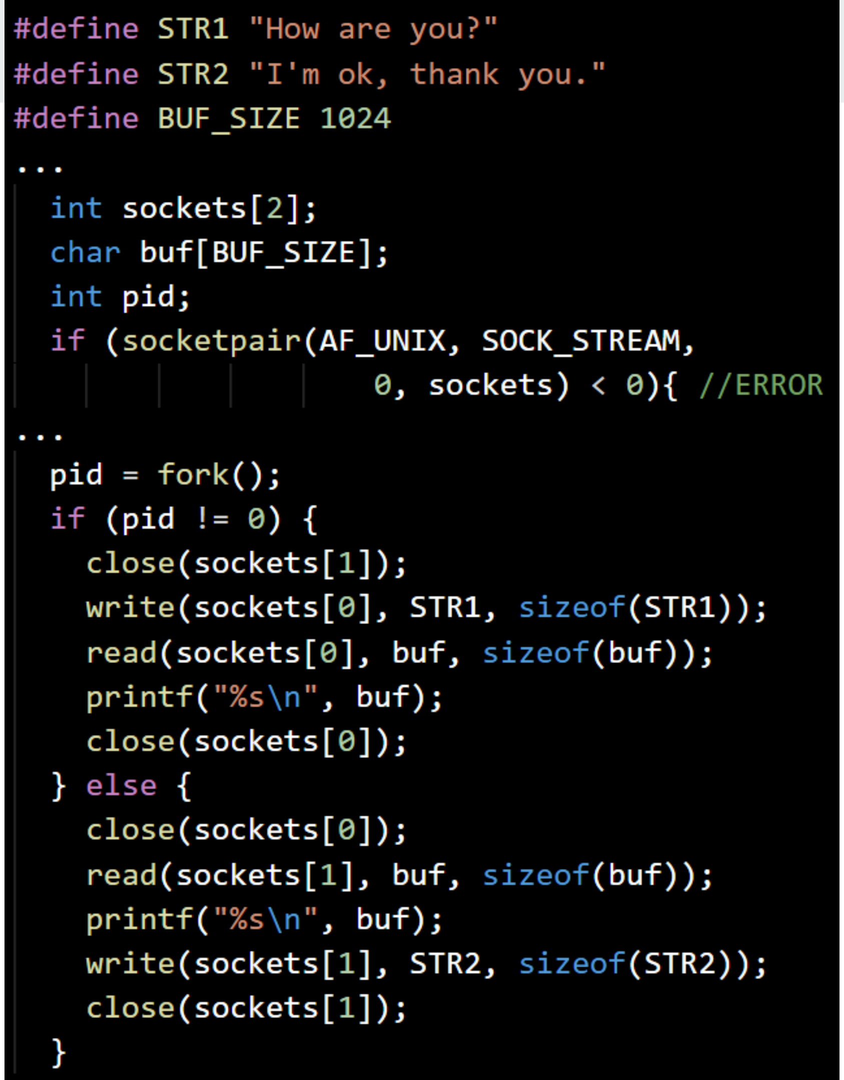
Write initial message to the “first” socket

Read answer from “first” socket, prit it and close the socket

<u>The child</u>: close the “first” socket

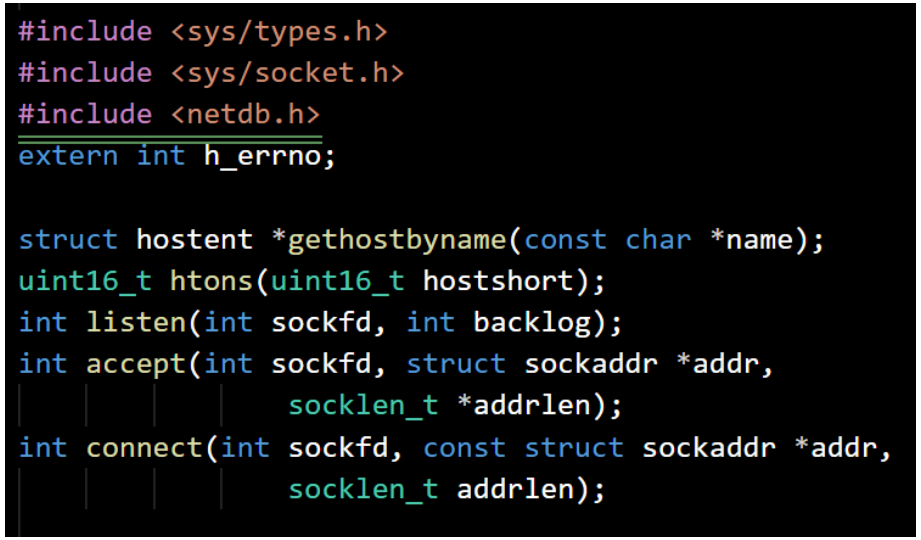
Read a message from the “second” socket and print it

Write answer to the “second” socket and close it



```
#define STR1 "How are you?"
#define STR2 "I'm ok, thank you."
#define BUF_SIZE 1024
...
  int sockets[2];
  char buf[BUF_SIZE];
  int pid;
  if (socketpair(AF_UNIX, SOCK_STREAM,
                  0, sockets) < 0){ //ERROR
...
  pid = fork();
  if (pid != 0) {
    close(sockets[1]);
    write(sockets[0], STR1, sizeof(STR1));
    read(sockets[0], buf, sizeof(buf));
    printf("%s\n", buf);
    close(sockets[0]);
  } else {
    close(sockets[0]);
    read(sockets[1], buf, sizeof(buf));
    printf("%s\n", buf);
    write(sockets[1], STR2, sizeof(STR2));
    close(sockets[1]);
  }
```

Network Sockets



```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);
uint16_t htons(uint16_t hostshort);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *addr,
                socklen_t *addrlen);
int connect(int sockfd, const struct sockaddr *addr,
                socklen_t addrlen);
```

To accept connections, the following steps are performed:

1. A socket is created with **socket**().
2. The socket is bound to a local address using **bind**(), so that other sockets may be **connect**()-ed to it.
3. A willingness to accept incoming connections and a queue limit for incoming connections are specified with **listen**().
4. Connections are accepted with **accept**().

netserver.c

File socket server

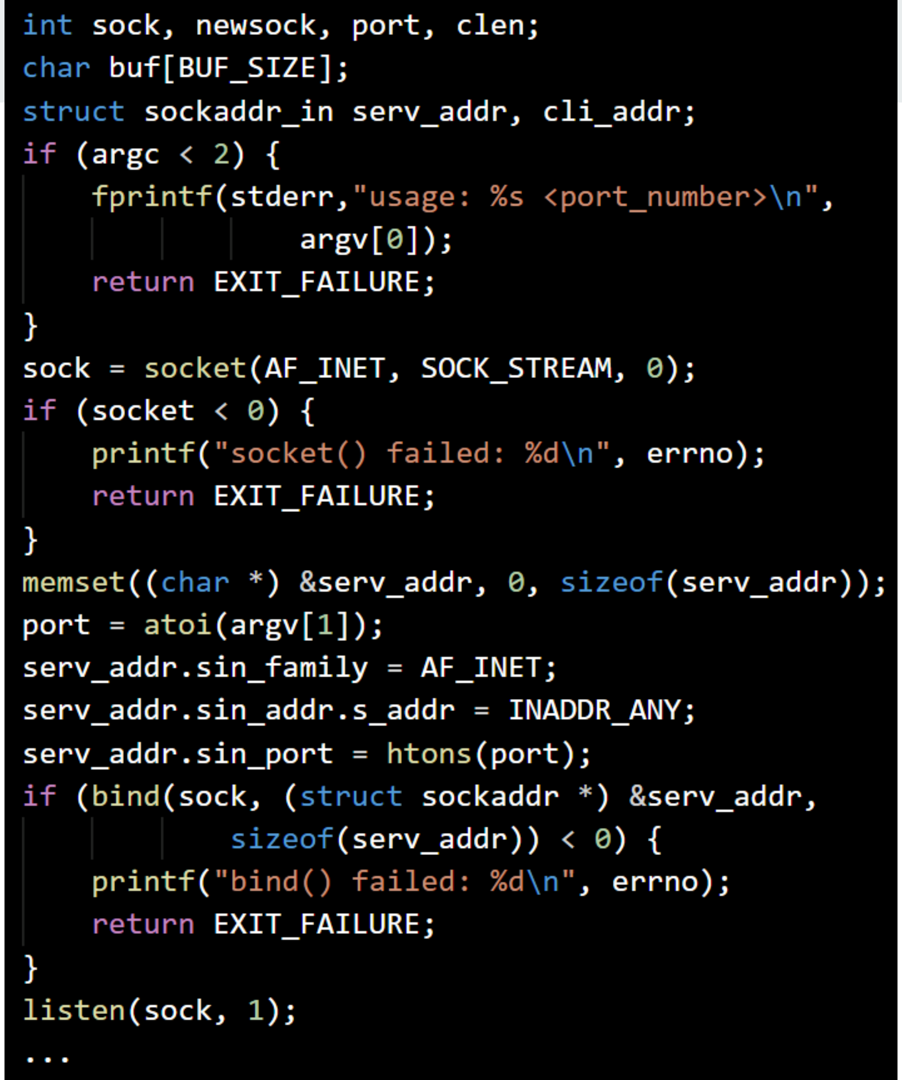
Check the input arguments, exit on error

Open **INET**, **STEAM** socket, exit on error

Initialize **sockaddr_in** structure for server

Bind the socket to the configured port, exit on error

Start listening, allow 1 connection



```c
int sock, newsock, port, clen;
char buf[BUF_SIZE];
struct sockaddr_in serv_addr, cli_addr;
if (argc < 2) {
    fprintf(stderr,"usage: %s <port_number>\n",
            argv[0]);
    return EXIT_FAILURE;
}
sock = socket(AF_INET, SOCK_STREAM, 0);
if (socket < 0) {
    printf("socket() failed: %d\n", errno);
    return EXIT_FAILURE;
}
memset((char *) &serv_addr, 0, sizeof(serv_addr));
port = atoi(argv[1]);
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(port);
if (bind(sock, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0) {
    printf("bind() failed: %d\n", errno);
    return EXIT_FAILURE;
}
listen(sock, 1);
...
```

netserver.c

File socket server (2)

Accept a new connection, exit on error

Read from the connection socket in buffer

Print the message from the buffer

Write ‘OK’ to the connection socket

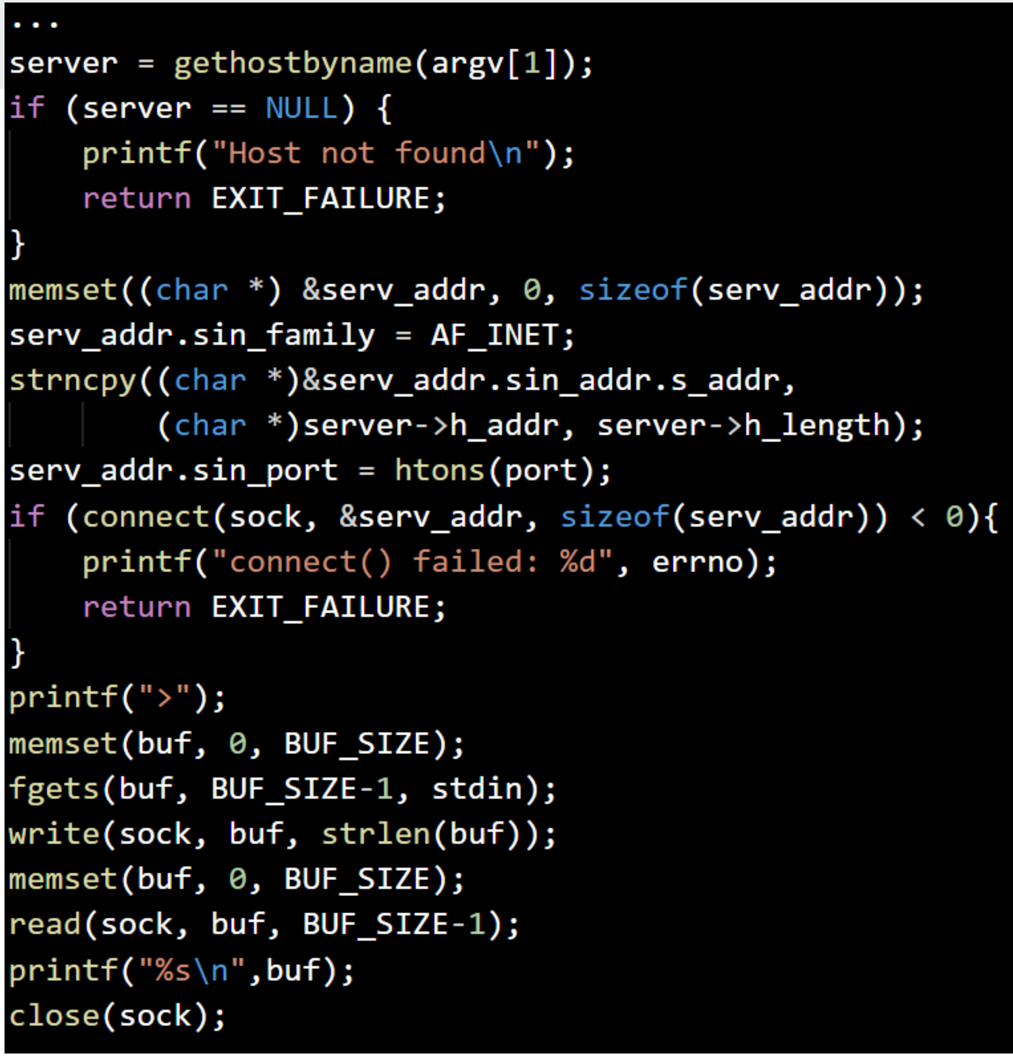
Close the connection and listening sockets

```c
...
clen = sizeof(cli_addr);
newsock = accept(sock,
                 (struct sockaddr *) &cli_addr,
                 &clen);
if (newsock < 0)
{
    printf("accept() failed: %d\n", errno);
    return EXIT_FAILURE;
}

memset(buf, 0, BUF_SIZE);
read(newsock, buf, BUF_SIZE-1);

buf[BUF_SIZE] = 0;
printf("MSG: %s\n", buf);
write(newsock, "OK", 2);

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

netclient.c

File socket client

- Check the input arguments, exit on error
- Open **INET**, **STEAM** socket, exit on error
- Get the server host, exit on error
- Configure and connect the socket to the server address and port, exit on error
- Indicate on the screen that input is expected and get a new text line
- Write the entered text to the socket
- Read an answer text from the socket, print it and close the socket



```
...
server = gethostbyname(argv[1]);
if (server == NULL) {
    printf("Host not found\n");
    return EXIT_FAILURE;
}
memset((char *) &serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
strncpy((char *)&serv_addr.sin_addr.s_addr,
        (char *)server->h_addr, server->h_length);
serv_addr.sin_port = htons(port);
if (connect(sock, &serv_addr, sizeof(serv_addr)) < 0){
    printf("connect() failed: %d", errno);
    return EXIT_FAILURE;
}
printf(">");
memset(buf, 0, BUF_SIZE);
fgets(buf, BUF_SIZE-1, stdin);
write(sock, buf, strlen(buf));
memset(buf, 0, BUF_SIZE);
read(sock, buf, BUF_SIZE-1);
printf("%s\n",buf);
close(sock);
```

Exercise

Project *BetterChat*:

Modify the network server/client pair of programs (‘**netserver.c**’ and ‘**netclient.c**’) from this lecture with the following improvements:

- Allow multiple messages conversation, until the client sends “**!quit**” to the server.
- The server should support multiple parallel clients connections.