**CSCD58H3 Winter 2018**

Tutorial: 003

Week: 4

Date: January 30th, 2018

---

# Socket Programming in C and Python

- ## What is a Socket

  A socket is a standard communication point or interface on the same device or different devices that connect an application to a network.

  **Families of Sockets**

  i. AF_INET:

  IPv4 Internet Protocols; the fourth version of the Internet Protocol. IP uses 32 bit addresses. In the format `xxx.xxx.xxx.xxx` where each `xxx` is a value between 0 and 255 inclusive. This family has only one protocol

  ii. AF_INET6:

  IPv6 Internet protocols. IP uses 128 bit addresses. In the format of eight groups of 2 bytes i.e `yyxx:yyxx:yyxx:yyxx:yyxx:yyxx:yyxx:yyxx` where each `yyxx` is a value from `0000` to `ffff` in hexadecimal. This family has only one protocol

  iii. AF_UNIX

  Also known as AF_LOCAL. Used for efficiently communicatin within the same machine. These can be unnamed, or bound to filesystem pathname that was created as a socket.

  iv. `AF_CAN`, `AF_IPX`, `AF_NETLINK`, `AF_X25`, `AF_AX25` etc.

  Supporting amateur radios, Kernel UI device, IPX, bluetooth, etc

  More information about family types can be found from `socket`man page.

  AF_INET6 is the family of protocol created to replace IPv4 since the addressing offered is very limited and millions of devices are getting connected to the internet. However, we will be demonstrating socket program with IPv4 (AF_INET)

  **Types of Sockets**

  i. STREAM Sockets: `SOCK_STREAM`

  One of the two most common socket types. Reliable, bidirectional data flow. Requires a valid connection. An out-of-band data transmission mechanism may be supported. The `read(...)` and `write(...)` or some variant e.g `send(...)` and `recv(...)` are typically used with this type fo socket. Think phone calls. More information `man 2 socket`

ii. DATAGRAM Sockets: `SOCK_DGRAM`

One of the two most common socket types. Unreliable, unidirectional data flow. No connection required. Packets; called datagrams, are usually received using `recvfrom(...)` and sent using `sendto(...)`. Think physical mail delivery. More information `man 2 socket`

iii. RAW sockets: `SOCK_RAW`

Only available to super user. Provides access to internal network protocol and interfaces. Usually useful if sending custom packets or building packets from scratch. More information available `man 7 raw`

- ## The Berkeley Sockets APIs

  Low level C Networking APIs available on most *NIX distro and on Windows via `Ws2_32.lib`. The Berkeley sockets API represents sockets as file descriptors.

  - `socket(...):` returns a file dscriptor to a socket given a specified socket family and type.
  - `bind(...):` binds a socket to an IP and Port given the socket's fd, an IP and a port
  - `connect(...):` connects to a given host and port using a socket fd
  - `listen(...):` listens for a specified number of connections to a bound socket, port and IP
  - `accept(...)` recieves connections to a bound socket, port and IP. Hangs until a connection is receives
  - `recv(...):` read or recv from a socket fd
  - `send(...):` write or send to a socket fd
  - `gethostbyaddr(...)`, `gethostbyname(...)`, `select(...)`etc
  - More information about the behaviour of these function can be obtained from their man pages (linked here).

  Data structures

  Most of the APIs listed above require some special structures in the C programming language. In this section, we investigate some of these structures and their correct initializations and usage. As there are two modes of IP addresses, namely IPv4: Using standard 32 bit IPs and IPv6 for 128 bits, we focus on IPv4 (AF_INET)

  - `struct in_addr`

    This struct represents the IP address and is defined in `netinet/in.h`

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

```
struct in_addr {
    unsigned long s_addr;  // set with inet_aton()
};
```

The IP address (`s_addr`) is specified using the API `inet_aton` (ASCII to Network) from `arpa/inet.h` which converts `x.x.x.x` IP format to Network-byte order.

```
// declare struct sockaddr_in myaddr
...
// specify internet addr
// myaddr.sin_addr <- is of type struct in_addr
inet_aton("127.0.0.1", &myaddr.sin_addr);
```

- `struct sockaddr_in`

  This struct is usually casted to `struct sockaddr` which is an equivalent sized struct but more generic. This allows different APIs to use different structs (same size) based on the socket family

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

  ```
  struct sockaddr_in {
      short            sin_family;   // socket family
      unsigned short   sin_port;     // port
      struct in_addr   sin_addr;     // IP addr; see struct above
      char             sin_zero[8];  // usually set to 0
  };

  struct sockaddr {
      sa_family_t sa_family;   // socket family
      char        sa_data[ ]; // generic data: maps to port, IP addr etc
                               // typically variable length, large
                               // enough to support any family
  }
  ```

  This struct represents the connection details i.e the family of IP address and port. It is defined in `netinet/in.h`

  The members `sin_family` and `sin_port` deserve special mention.

  - `sin_family` specifies the family of the socket type. We will investigate this more when learning to create sockets.
  - `sin_port` corresponds to the port number the socket will be bound to. However, it must be set in Network-byte order like IP addresses. The API `htons(...)`(Host to

network short) from `arpa/inet.h` performs this conversion.

```
// declare struct sockaddr_in myaddr
myaddr.sin_port = htons(8080);
...
```

- **Creating sockets in C and Python**

  We provide two language support, you can follow the explanation in the language of your choice but keep in mind, you will be programming your assignments in C

  - C

    ```
    int socket(int socket_family, int socket_type, int protocol);
    ```

    - RETURN TYPE `int`: This API returns a socket file descriptor on success and -1 on failure. Must always check the return value is not -1.
    - `int protocol`: This parameter specifies the protocol to use with the family of sockets. However, most families; or atleast the ones we are concerned with have only one protocol. Hence, the value of this parameter is mostly always 0.
    - `int socket_family` and `int socket_type` correspond to the socket families and types discussed earlier. e.g `AF_INET` and `SOCK_STREAM`. These macros are defined in `sys/socket.h`

    Example:

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

      ...
      // creating an IPv4 streaming socket
      int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
      if (socket_fd == -1) {
          // handle error and quit gracefully

      }
      ```

  - **Python**

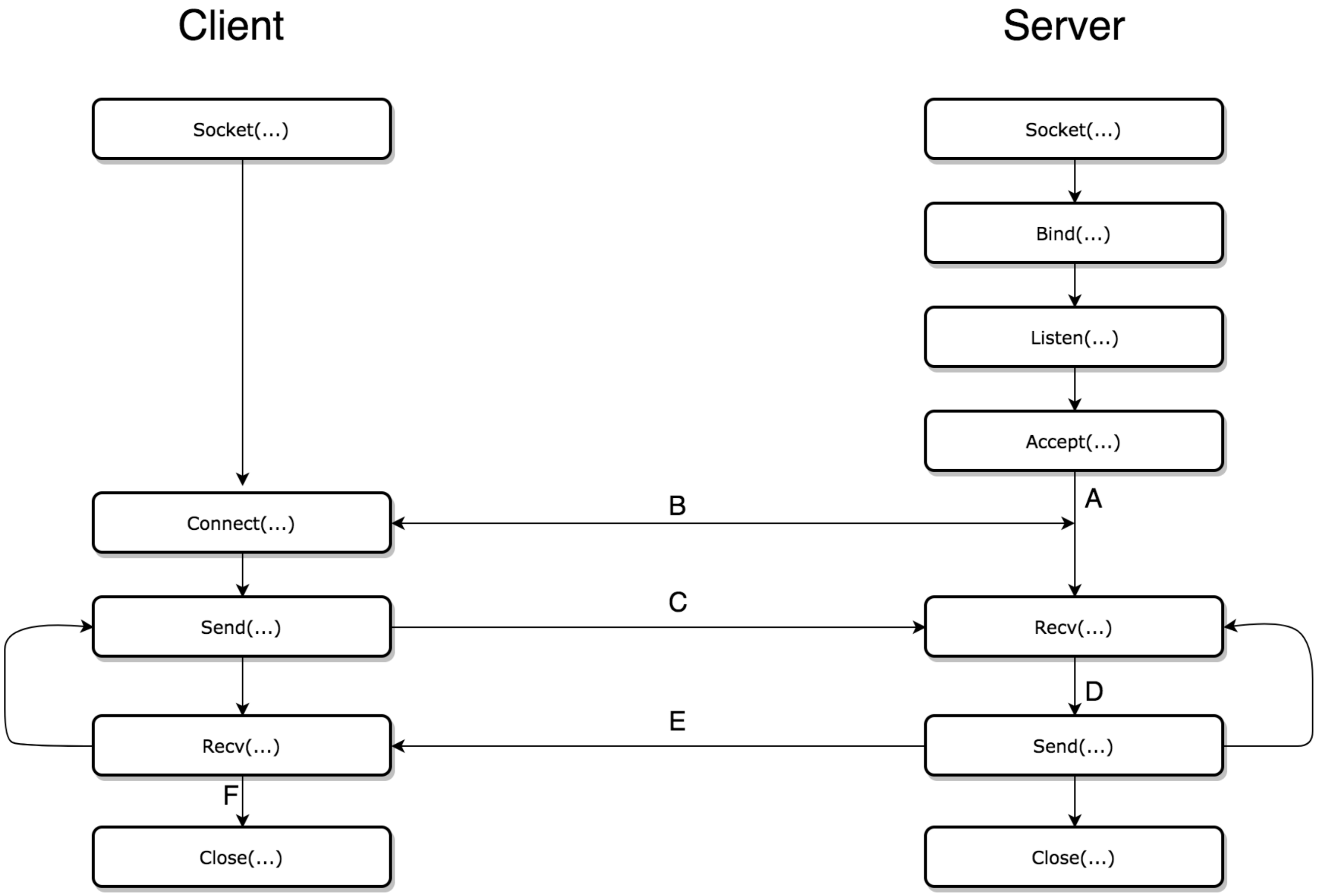
    ```
    socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)
    ```

    Parameters correspond to the those explained above. By default, the `socket.socket()` creates an IPv4 streaming socket.

    Example:

```
import socket

# creating an IPv4 streaming socket
sock_fd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
if sock_fd == -1:
   # handle error and quit gracefully

...
```

- **The TCP Client-Server Model for Socket Programming**



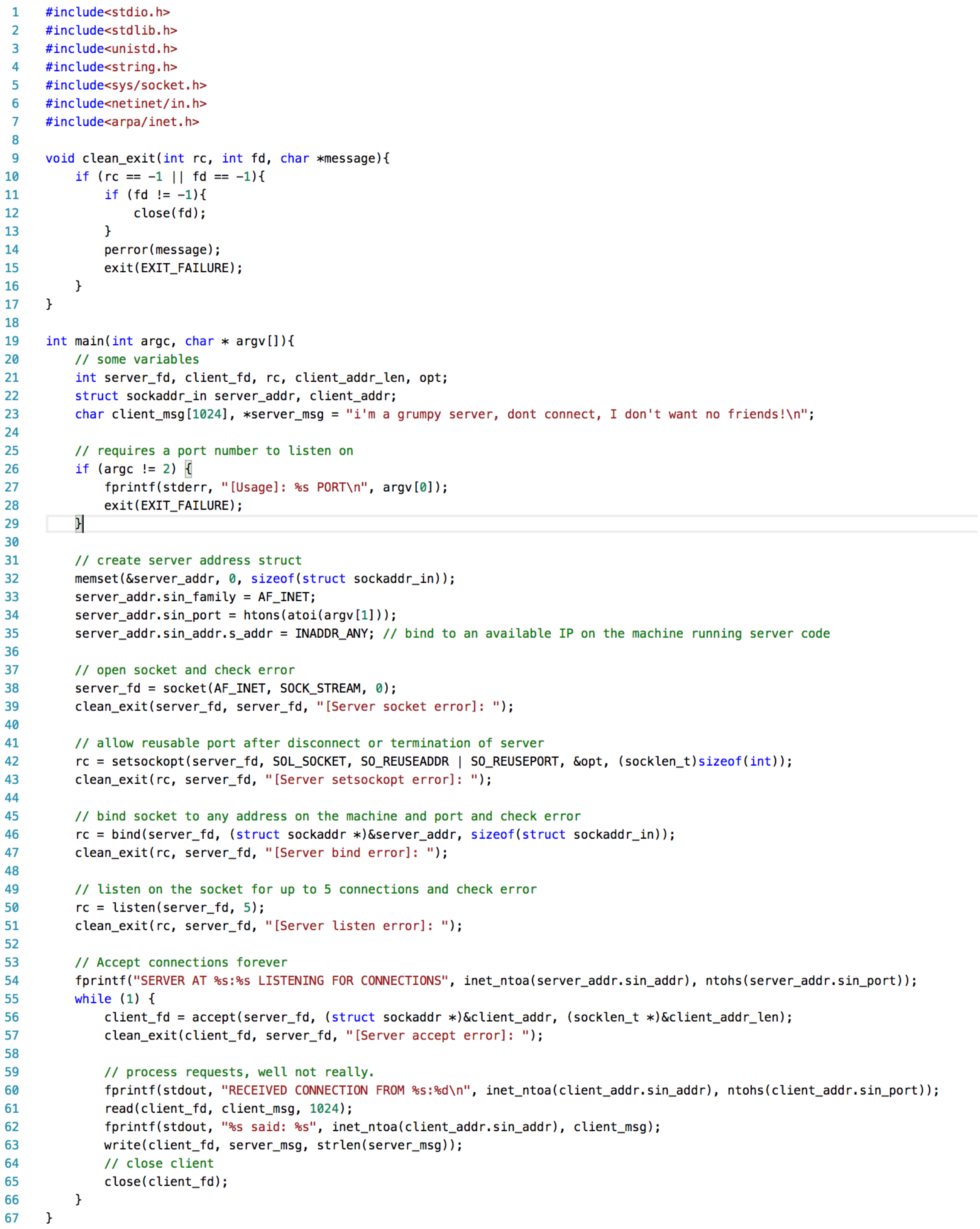
  - A: `accept(...)` blocks until it receives a connection
  - B: TCP handshake, connection established
  - C: Client's request
  - D: Server processes Client's request
  - E: Server's response
  - C -> D -> E loops until client is done sending requests

- F: Client closes connection to Server

- ## Simple Server Programming

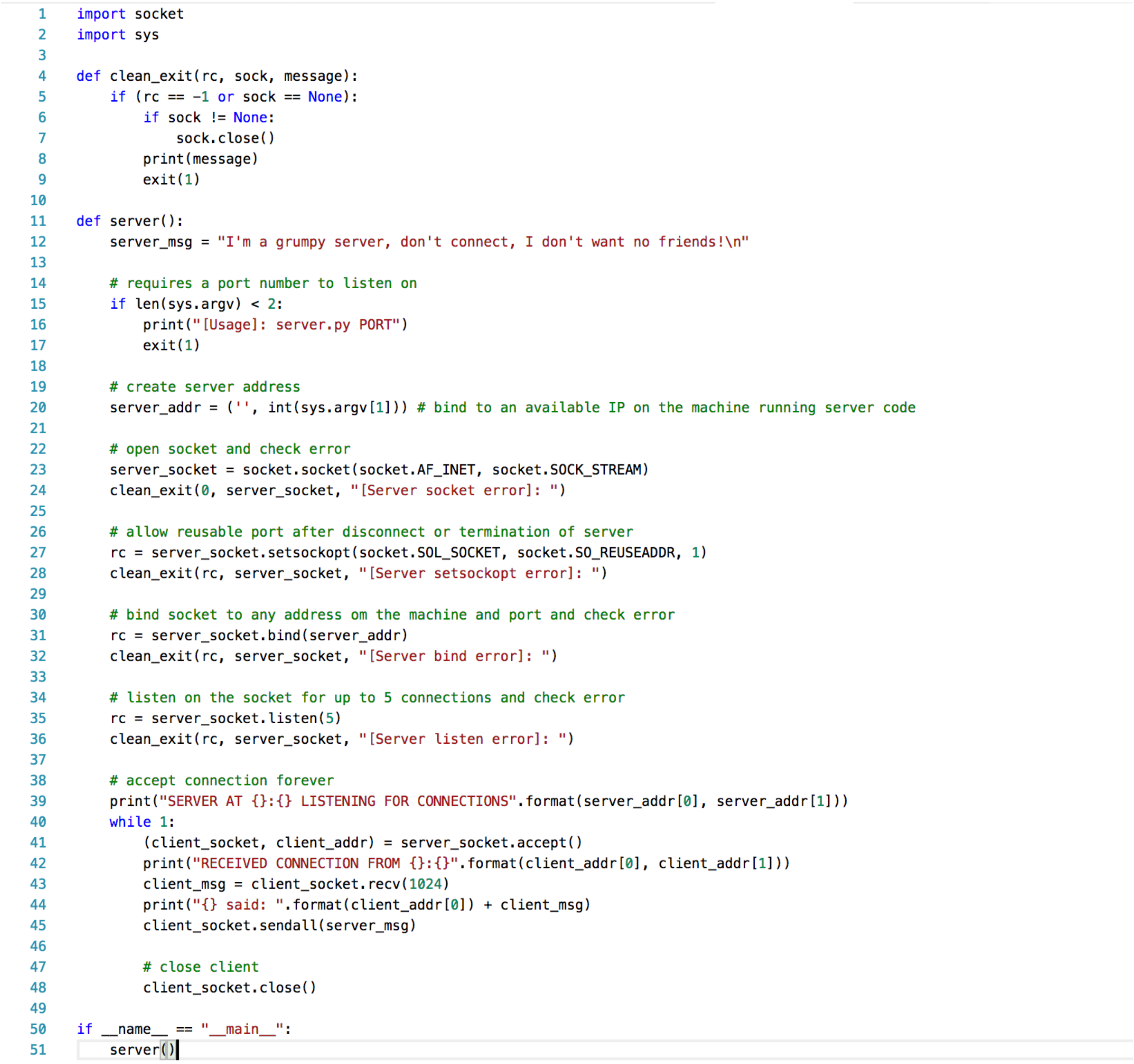
  Using `nc` or custom simple client as client

  - ### C walkthrough



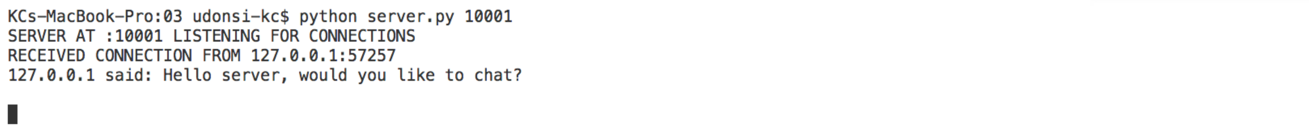
```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>

void clean_exit(int rc, int fd, char *message){
    if (rc == -1 || fd == -1){
        if (fd != -1){
            close(fd);
        }
        perror(message);
        exit(EXIT_FAILURE);
    }
}

int main(int argc, char * argv[]){
    // some variables
    int server_fd, client_fd, rc, client_addr_len, opt;
    struct sockaddr_in server_addr, client_addr;
    char client_msg[1024], *server_msg = "i'm a grumpy server, dont connect, I don't want no friends!\n";

    // requires a port number to listen on
    if (argc != 2) {
        fprintf(stderr, "[Usage]: %s PORT\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // create server address struct
    memset(&server_addr, 0, sizeof(struct sockaddr_in));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(atoi(argv[1]));
    server_addr.sin_addr.s_addr = INADDR_ANY; // bind to an available IP on the machine running server code

    // open socket and check error
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    clean_exit(server_fd, server_fd, "[Server socket error]: ");

    // allow reusable port after disconnect or termination of server
    rc = setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, (socklen_t)sizeof(int));
    clean_exit(rc, server_fd, "[Server setsockopt error]: ");

    // bind socket to any address on the machine and port and check error
    rc = bind(server_fd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr_in));
    clean_exit(rc, server_fd, "[Server bind error]: ");

    // listen on the socket for up to 5 connections and check error
    rc = listen(server_fd, 5);
    clean_exit(rc, server_fd, "[Server listen error]: ");

    // Accept connections forever
    fprintf("SERVER AT %s:%s LISTENING FOR CONNECTIONS", inet_ntoa(server_addr.sin_addr), ntohs(server_addr.sin_port));
    while (1) {
        client_fd = accept(server_fd, (struct sockaddr *)&client_addr, (socklen_t *)&client_addr_len);
        clean_exit(client_fd, server_fd, "[Server accept error]: ");

        // process requests, well not really.
        fprintf(stdout, "RECEIVED CONNECTION FROM %s:%d\n", inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
        read(client_fd, client_msg, 1024);
        fprintf(stdout, "%s said: %s", inet_ntoa(client_addr.sin_addr), client_msg);
        write(client_fd, server_msg, strlen(server_msg));
        // close client
        close(client_fd);
    }
}
```

- ### Python walkthrough

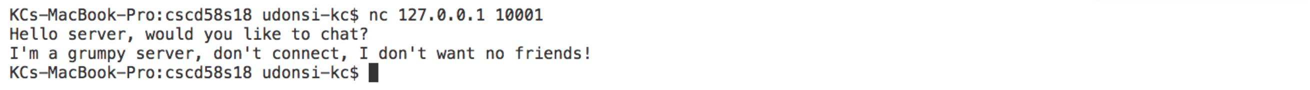


```python
import socket
import sys

def clean_exit(rc, sock, message):
    if (rc == -1 or sock == None):
        if sock != None:
            sock.close()
        print(message)
        exit(1)

def server():
    server_msg = "I'm a grumpy server, don't connect, I don't want no friends!\n"

    # requires a port number to listen on
    if len(sys.argv) < 2:
        print("[Usage]: server.py PORT")
        exit(1)

    # create server address
    server_addr = ('', int(sys.argv[1])) # bind to an available IP on the machine running server code

    # open socket and check error
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    clean_exit(0, server_socket, "[Server socket error]: ")

    # allow reusable port after disconnect or termination of server
    rc = server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    clean_exit(rc, server_socket, "[Server setsockopt error]: ")

    # bind socket to any address om the machine and port and check error
    rc = server_socket.bind(server_addr)
    clean_exit(rc, server_socket, "[Server bind error]: ")

    # listen on the socket for up to 5 connections and check error
    rc = server_socket.listen(5)
    clean_exit(rc, server_socket, "[Server listen error]: ")

    # accept connection forever
    print("SERVER AT {}:{} LISTENING FOR CONNECTIONS".format(server_addr[0], server_addr[1]))
    while 1:
        (client_socket, client_addr) = server_socket.accept()
        print("RECEIVED CONNECTION FROM {}:{}".format(client_addr[0], client_addr[1]))
        client_msg = client_socket.recv(1024)
        print("{} said: ".format(client_addr[0]) + client_msg)
        client_socket.sendall(server_msg)

        # close client
        client_socket.close()

if __name__ == "__main__":
    server()
```

- ### Communication with `nc`



```
KCs-MacBook-Pro:03 udonsi-kc$ python server.py 10001
SERVER AT :10001 LISTENING FOR CONNECTIONS
RECEIVED CONNECTION FROM 127.0.0.1:57257
127.0.0.1 said: Hello server, would you like to chat?

█
```

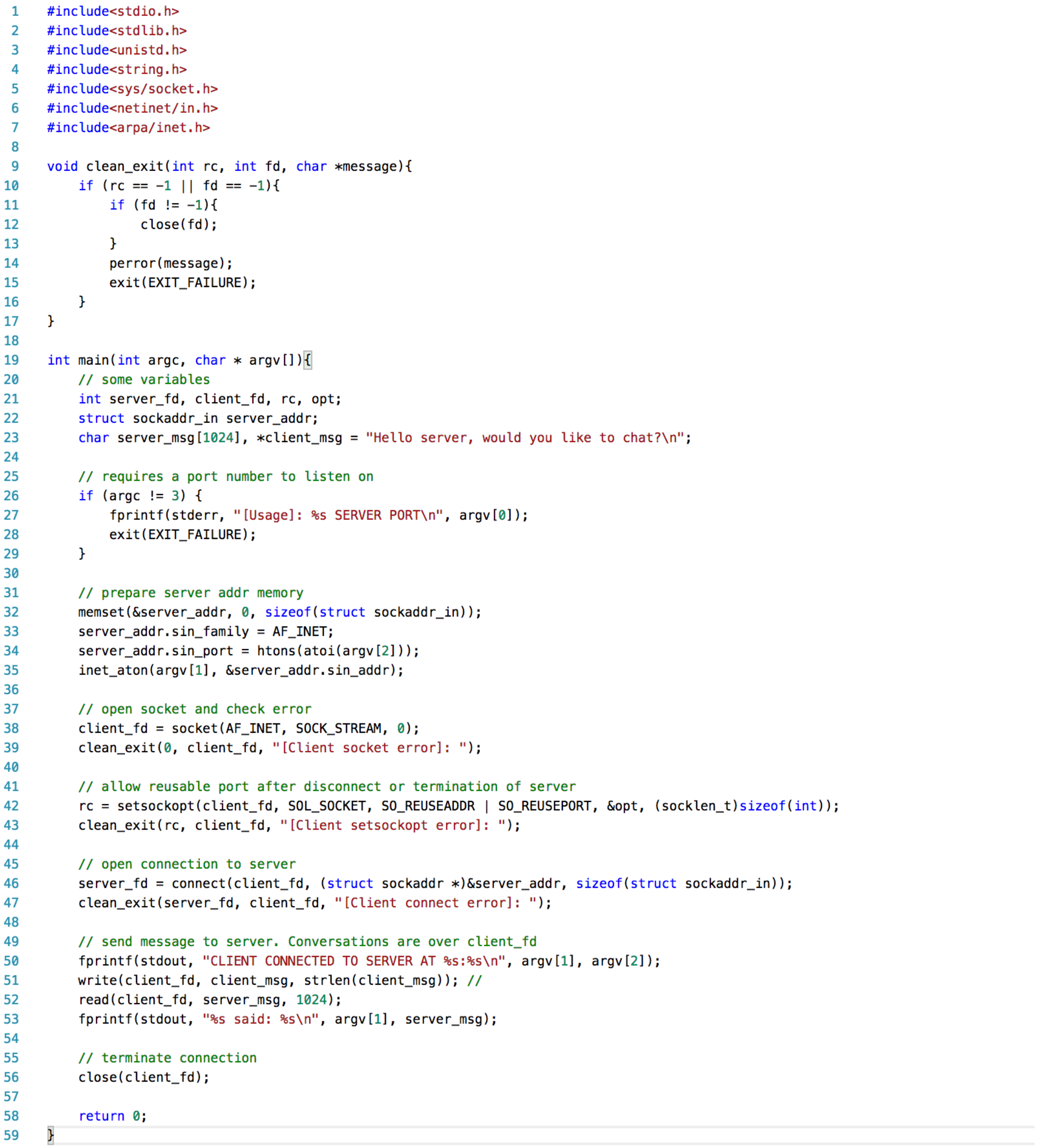


```
KCs-MacBook-Pro:cscd58s18 udonsi-kc$ nc 127.0.0.1 10001
Hello server, would you like to chat?
I'm a grumpy server, don't connect, I don't want no friends!
KCs-MacBook-Pro:cscd58s18 udonsi-kc$ █
```

## Simple Client Programming

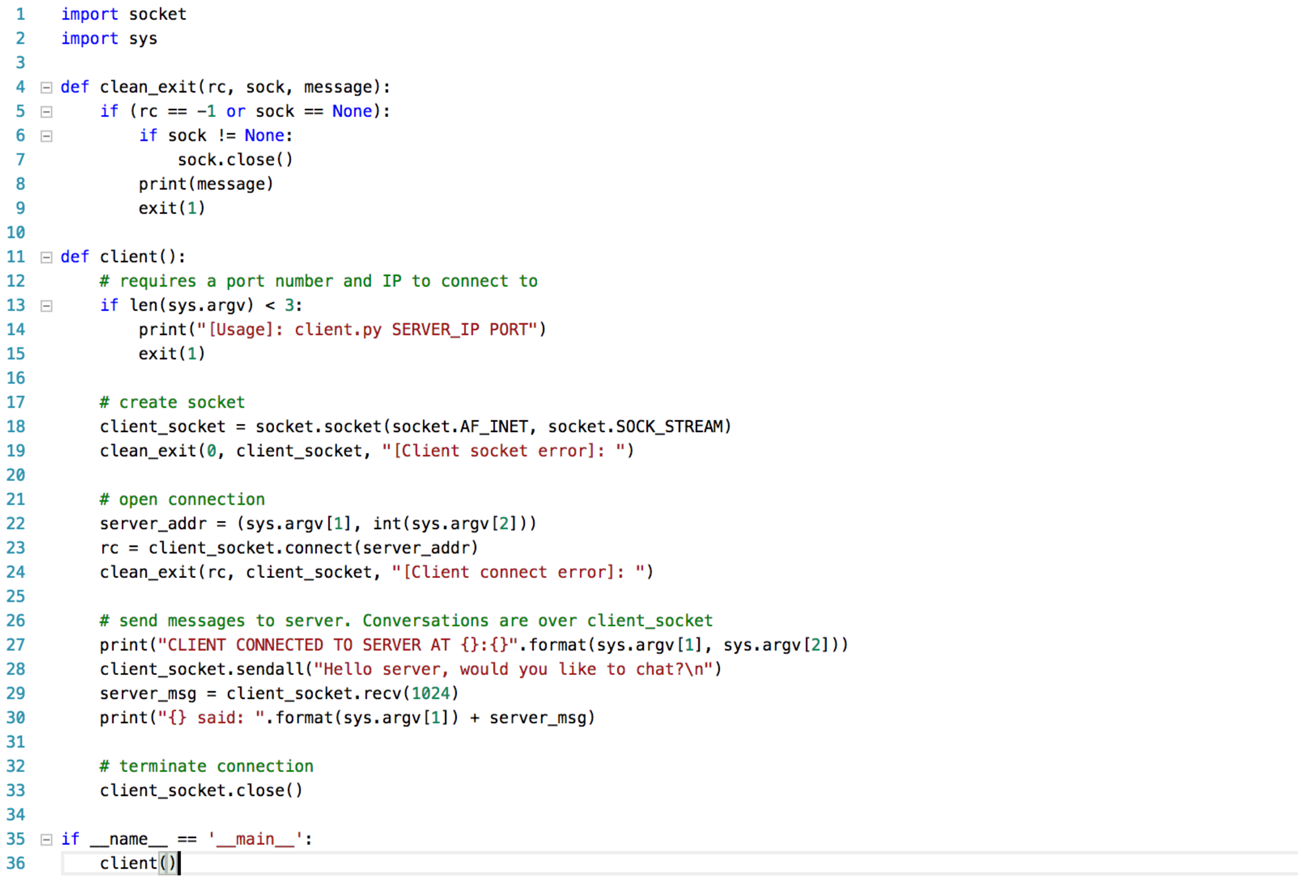
Using `nc` or custom simple server as server

- ### C walkthrough



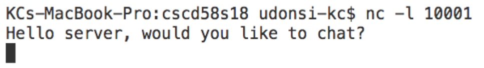
```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>

void clean_exit(int rc, int fd, char *message){
    if (rc == -1 || fd == -1){
        if (fd != -1){
            close(fd);
        }
        perror(message);
        exit(EXIT_FAILURE);
    }
}

int main(int argc, char * argv[]){
    // some variables
    int server_fd, client_fd, rc, opt;
    struct sockaddr_in server_addr;
    char server_msg[1024], *client_msg = "Hello server, would you like to chat?\n";

    // requires a port number to listen on
    if (argc != 3) {
        fprintf(stderr, "[Usage]: %s SERVER PORT\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // prepare server addr memory
    memset(&server_addr, 0, sizeof(struct sockaddr_in));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(atoi(argv[2]));
    inet_aton(argv[1], &server_addr.sin_addr);

    // open socket and check error
    client_fd = socket(AF_INET, SOCK_STREAM, 0);
    clean_exit(0, client_fd, "[Client socket error]: ");

    // allow reusable port after disconnect or termination of server
    rc = setsockopt(client_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, (socklen_t)sizeof(int));
    clean_exit(rc, client_fd, "[Client setsockopt error]: ");

    // open connection to server
    server_fd = connect(client_fd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr_in));
    clean_exit(server_fd, client_fd, "[Client connect error]: ");

    // send message to server. Conversations are over client_fd
    fprintf(stdout, "CLIENT CONNECTED TO SERVER AT %s:%s\n", argv[1], argv[2]);
    write(client_fd, client_msg, strlen(client_msg)); //
    read(client_fd, server_msg, 1024);
    fprintf(stdout, "%s said: %s\n", argv[1], server_msg);

    // terminate connection
    close(client_fd);

    return 0;
}
```

- ## Python walkthrough

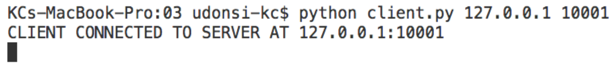


```python
import socket
import sys

def clean_exit(rc, sock, message):
    if (rc == -1 or sock == None):
        if sock != None:
            sock.close()
        print(message)
        exit(1)

def client():
    # requires a port number and IP to connect to
    if len(sys.argv) < 3:
        print("[Usage]: client.py SERVER_IP PORT")
        exit(1)

    # create socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    clean_exit(0, client_socket, "[Client socket error]: ")

    # open connection
    server_addr = (sys.argv[1], int(sys.argv[2]))
    rc = client_socket.connect(server_addr)
    clean_exit(rc, client_socket, "[Client connect error]: ")

    # send messages to server. Conversations are over client_socket
    print("CLIENT CONNECTED TO SERVER AT {}:{}".format(sys.argv[1], sys.argv[2]))
    client_socket.sendall("Hello server, would you like to chat?\n")
    server_msg = client_socket.recv(1024)
    print("{} said: ".format(sys.argv[1]) + server_msg)

    # terminate connection
    client_socket.close()

if __name__ == '__main__':
    client()
```

- ### Communication with `nc`



```
KCs-MacBook-Pro:cscd58s18 udonsi-kc$ nc -l 10001
Hello server, would you like to chat?
█
```



```
KCs-MacBook-Pro:03 udonsi-kc$ python client.py 127.0.0.1 10001
CLIENT CONNECTED TO SERVER AT 127.0.0.1:10001
█
```

## References and Resources

0. man pages and the good ol' commandline :smile:
1. wikipedia
2. University of Glasgow: Network programming in C