Socket Programing
In Linux - C

Courtesy of Roni Levin, Yoad Zur & Chen Griner

Introduction

Why learn about sockets?

- The basics of app-to-app communication over the internet.

What is the goal of this presentation?

- **Introduce** you with some concepts of socket programming. “ How to work with network sockets? “

This is a very wide subject

- This is not a complete guide! You are expected to explore further by yourself.

Today’s Presentation:

1

- High-level Overview, clients and servers

2

- Familiarize yourself with socket API

3

- A simple socket connection scheme

5

- Socket management

6

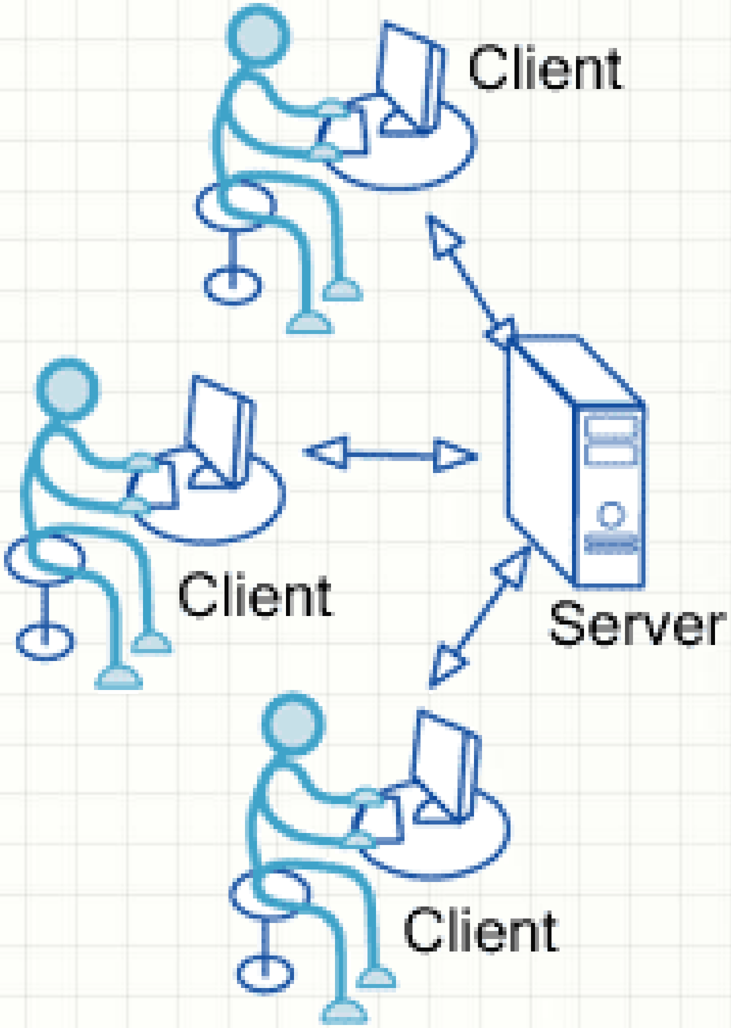
- Lab Session

HIGH-LEVEL OVERVIEW: CLIENTS AND SERVERS

Client – Server

- **Clients**:
  - Locates the server
  - Initiate connection
  - Example: you.

- **Server**:
  - Responder.
  - Provides service.
  - Example: Moodle.



Client – *Server:* some key differences

Clients	Server
□ Simple	□ Complex
□ (Usually) sequential	□ (Massively) concurrent
□ Not performance sensitive	□ High performance
□ **Execute on-demand**	□ **Always-on**

Client – *Server*: Similarities

- ❑ Share common protocols
  - o Network layer
  - o Transport layer
  - o Application layer
- ❑ Both rely on APIs for network access

What is an API?

application programming interface (**API**) is a set of routines, protocols, and tools for building software applications. For example network sockets

What is a network socket?

It is an application’s “mailbox” for network messages.

Used to pass messages among applications on different computers.

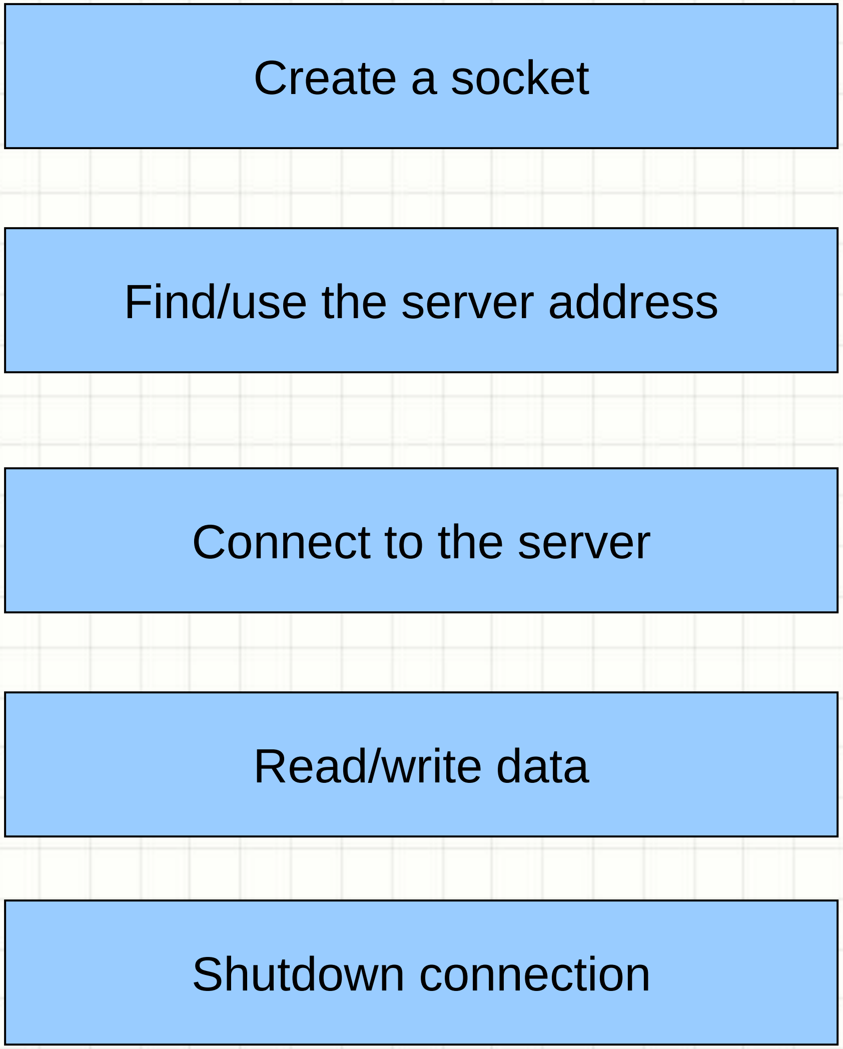
Managed by the operating system.

Represented as a *“file descriptor”*.

Implements an Incoming and Outgoing queues.

Identified by IP address, port and protocol.

Client: general workflow



Create a socket

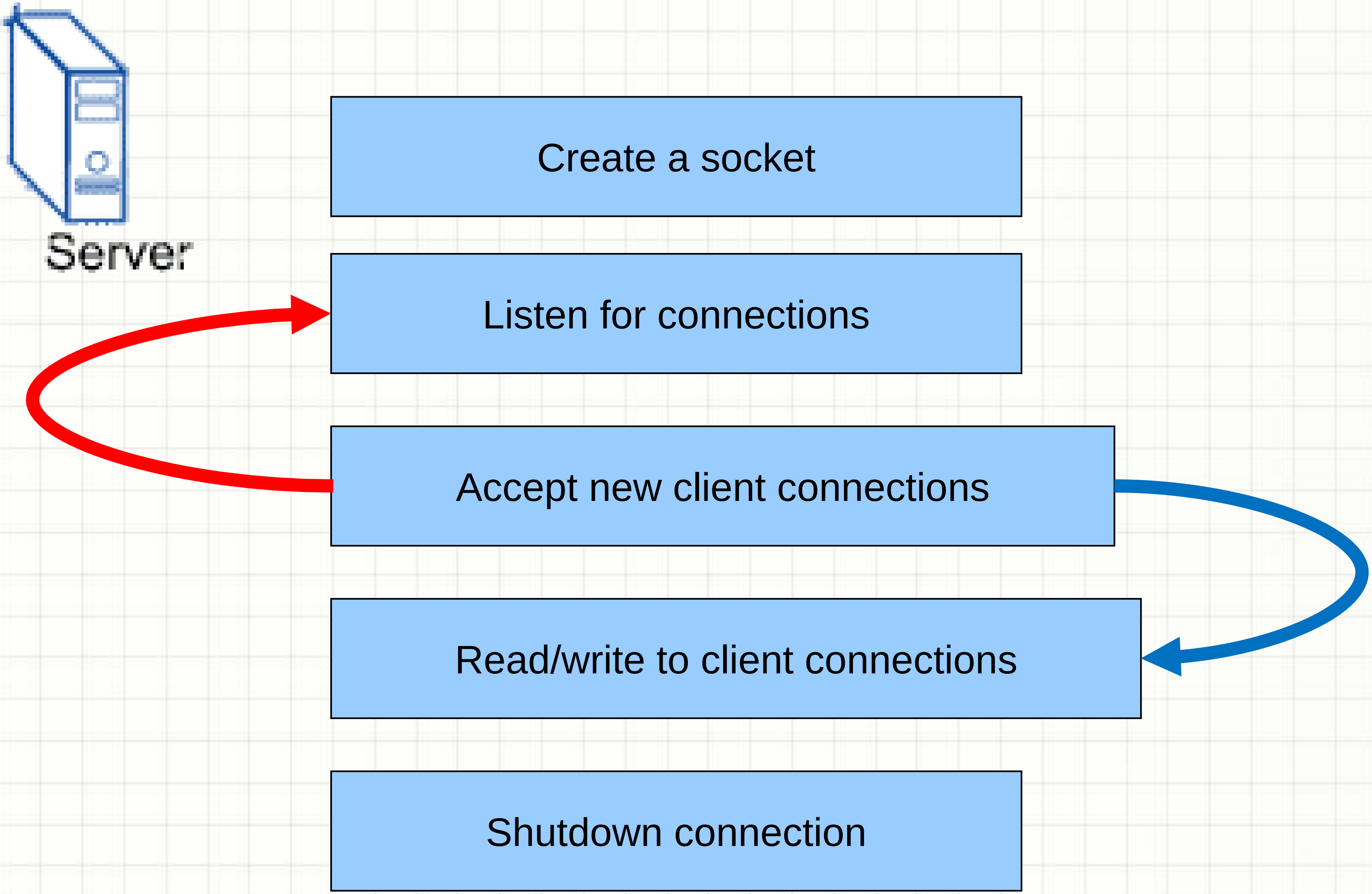
Find/use the server address

Connect to the server

Read/write data

Shutdown connection

Server: *general work flow*



Create a socket

Listen for connections

Accept new client connections

Read/write to client connections

Shutdown connection

TCP Server:

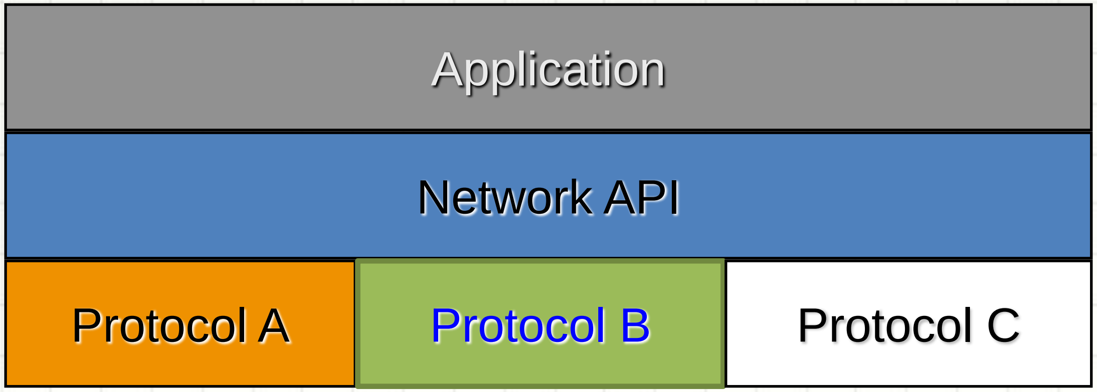
The “Welcome socket”

- The client needs a known address and port to connect to.
- A TCP server has at least one special socket we call the “welcome socket”.
- First connection by the client is to this socket.
- New separate socket for each new client.
- The “welcome socket” is permanent.
  - It is not closed when the client leaves.
- The welcome socket has a queue.
  - Can’t have too many clients entering at the same time.

Socket API

Introduction

- A **socket API** is an application programming interface (API), that allows applications to control and use network sockets.
- Sockets provide the interface between application and the protocol software.



Application

Network API

Protocol A

Protocol B

Protocol C

Berkeley sockets / POSIX sockets

- Used in Unix systems since 1983.
- We will run our programs in Linux.
- An API for Internet sockets.
  - Essentially, you get a set of C functions.
  - Each function has a different role in the operation of sockets.
  - What functions to use?

Where do we start?

Basic example, opening a socket

- Let’s first declare two sockets, one for UDP and one for TCP.
  - (1) A socket is a file-descriptor. An integer.
- Let’s tell the OS we want to make a new socket.
  - (2) we use the socket() function and get a TCP socket.
  - (3) same for UDP.
- SOCK_STREAM Or SOCK_DGRAM determine the socket type.
- Is that it?
  - There are many other parts.

```
1.   int myTCPsocket, myUDPsocket;
2.  myTCPsocket = socket(AF_INET, SOCK_STREAM, 0);
3.  myUDPsocket =socket(AF_INET,SOCK_DGRAM,0));
```

Socket Types

- The stream socket
  - Data should be handled as a part of a stream
  - Maintain a constant connection
  - Uses TCP to set up a reliable connection
  - Use: SOCK_STREAM
- The datagram socket.
  - Data is separate packets.
  - Use UDP
  - Unreliable
  - Use: SOCK_DGRAM
- Another
  - Raw socket
  - (don't) Use: SOCK_RAW

Socket API

- socket(): creates a socket of a given domain, type, protocol
- http://linux.die.net/man/2/socket

1. TCP: socket(AF_INET, SOCK_STREAM, 0)
2. UDP: socket(AF_INET, SOCK_DGRAM, 0)

- bind(): a socket start with no address, assigns an address to the socket.
  - Remember to fill and cast an appropriate “sockaddr” type struct.
- http://linux.die.net/man/2/bind

1. TCP server & UDP: bind(lisen_sock, (struct sockaddr *)&listen_addr, sizeof(listen_addr)

Ipv4 socket address structure

```
struct socketaddr_in{
  uint8_t          sin_len;    /*length of the structure (16)*/
  sa_falimily_t    sin_family /* AF_INT*/
  in_port_t        sin_port    /* 16 bit TCP or UDP port number*/
  struct in_addr   sin_addr    /* 32 bit Ipv4 address */
  char             sin_zero(8)/* unused*/
}
```

Socket API

- listen(): specifies the number of pending concurrent connections that can be queued for a server socket.
  - Makes a “welcome socket”
- http://linux.die.net/man/2/listen

1. TCP: listen(lisen_sock, SOMAXCONN)

Socket API

- accept(): server accepts a connection request from a client. Blocking until a connection is received.
- http://linux.die.net/man/2/accept

```
1.    TCP: accept(lisen_sock, (struct  sockaddr*) &client_addr,  &client_addr_size)
```

- inet_addr(): converts the Internet host address *cp* from IPv4 numbers-and-dots notation into binary data in network byte order
  - Another similar function gethostbyname().
- https://linux.die.net/man/3/inet_addr

```
1.    TCP or UDP: in_addr_t inet_addr(const char *cp);
```

- connect(): client requests a connection request to a server (call)
- http://linux.die.net/man/2/connect

```
1.    TCP: connect(server_sock, (struct  sockaddr*) &server_addr, sizeof(server_addr))
```

Socket API

- recv(), recvfrom(): read from a connection
  - Positive returned value marks the number of successfully received bytes
  - negative returned value means an error.
  - 0 means the connection has been closed.
    - On which protocol is this relevant?
- http://linux.die.net/man/2/recv

1. TCP & UDP : recv(sock, buffer, &buff_len, 0)

- send(), sendto(): write to a connection
  - Positive returned value marks the number of sent bytes.
    - What happens if not all bytes were sent?
  - negative returned value means an error.
- http://linux.die.net/man/2/send

1. TCP & UDP : send(sock, buffer, &buff_len, 0)

Socket API

- Setsockopt(…): assigns options to the socket.
  - Join/leave multicast group.
  - Other options.
  - Use as necessary.
- https://linux.die.net/man/2/setsockopt

TCP & UDP :

```
int setsockopt(int socket, int level, int option_name, const void *option_value, socklen_t option_len);
```

Socket API

- Shutdown(…). shut down socket send and receive operations.
  - A socket is full duplex.
  - Stop receiving or stop sending.
  - You may just use close(..)
- https://linux.die.net/man/3/shutdown

```
TCP & UDP :
int shutdown(int socket, int how);
```

- Close(…): close the socket.
  - What happens when you close a TCP/UDP socket?
- http://linux.die.net/man/2/close

```
1.   TCP & UDP : close(socket)
```

Socket API

- htons(), htonl(), ntohs(), ntohl(): Convert to or from host byte order to network byte order, i.e., big endian⇔ little endian.
  - htons(): host to network short.
  - htonl(): host to network long.
  - ntohs(): network to host short.
  - ntohl(): network to host long.
- http://linux.die.net/man/3/htons

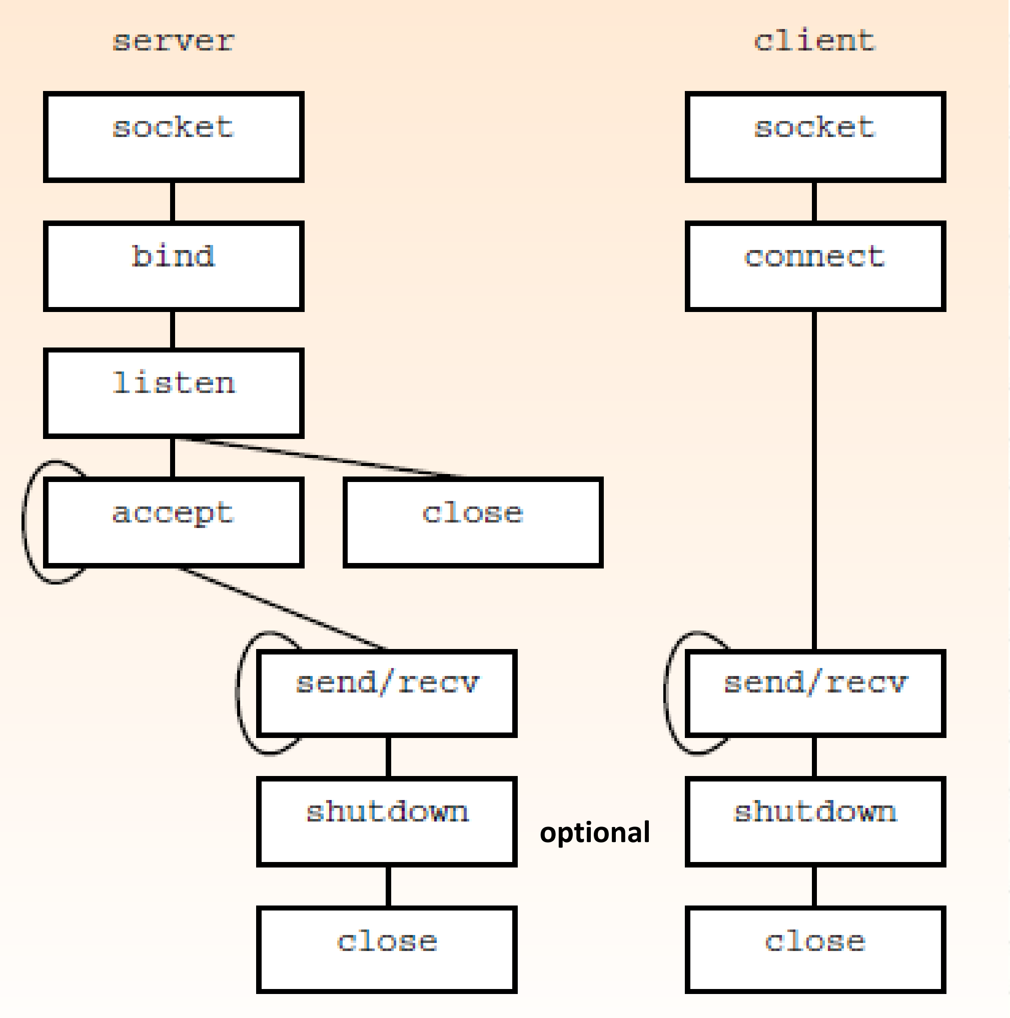
1. TCP & UDP : htons(1234);//1234=port number

There are many more socket related functions

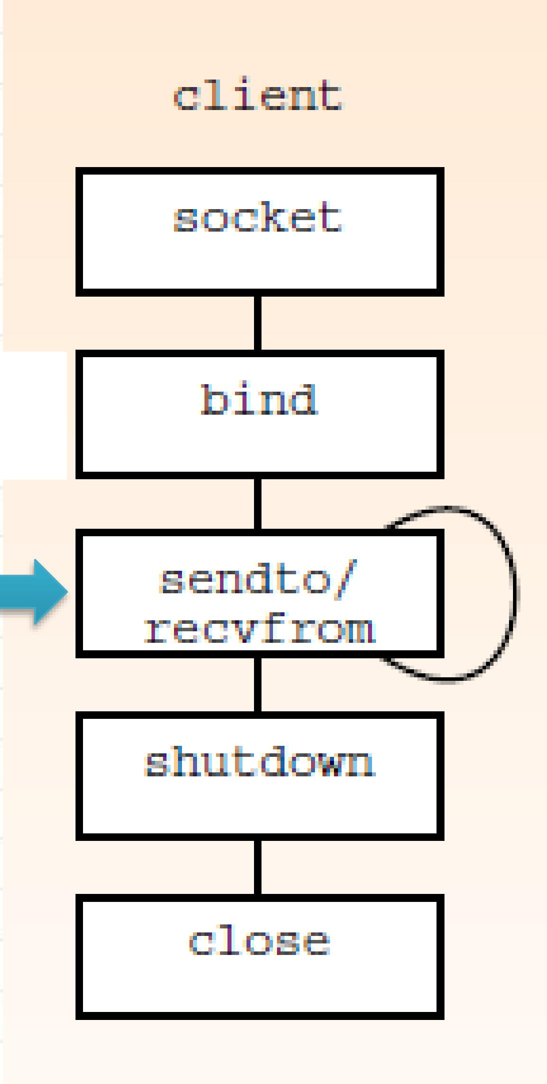
- Extra: a nice example of client-server program server using socket API:
- https://www.programminglogic.com/example-of-client-server-program-in-c-using-sockets-and-tcp/

Socket connection scheme

TCP


server
client
socket
socket
bind
connect
listen
accept
close
send/recv
send/recv
shutdown
optional
shutdown
close
close

UDP



Is bind always necessary?

Send/Receive

Socket management

Socket management methods

- Servers need to manage many socket simultaneously.
  - At least one “welcome socket”.
  - Many clients.
  - How?
- Parallel: Threads / Process.

- Serial: select() function

Serial Socket management: Select()

- Select(): waits for sockets(or any FD) to change status
  - To use select() first define a set of FDs (fd_set).
    - readfds :a set of *read* descriptors to monitor .
    - writefds :a set of *write* descriptors to monitor
- Select() start blocking until an event occurs.
  - An event could be an incoming message, a timeout and more.
- http://linux.die.net/man/2/select
- Read more about select: http://www.lowtek.com/sockets/select.html

TCP & UDP : int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

```
void FD_CLR(int fd, fd_set *set);
int FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

Select(); Example

The following is a single use case for select().

We listen for user input until a time out has occurred or the user has entered some input.

Note that this example is incomplete.

```
TCP & UDP & files
fd_set fdset;// set of file dicriptors
FD_ZERO(& fdset); // clear the set
FD_SET(fileno(stdin), & fdset);//set stdin as a  file descriptor in the set.
timeout =set_timeout(60 sec)//set the timeout for select, not a “real” function.
inputfd=select(FD_SETSIZE,&fdset, NULL , NULL,&timeout)//block until timeout, or user input
        if(inputfd==0)
                {puts(“Opps, Time Out!!!”)}//not a “real” function
        else if(FD_ISSET(fileno(stdin),& fdset)
                {user_handler()} //not a “real” function
```

Select(); Example

Some questions?

- Do we have to use a timeout?
  - No.
- Can we listen to multiple sockets using the same FD_set?
  - Yes.
- Can we have more than one file descriptor “jumping” in the same time.
  - Yes.
- Will this select() work multiple times?
  - Not without some further work.

To understand how to work with select() you will have to do some further research on your own.

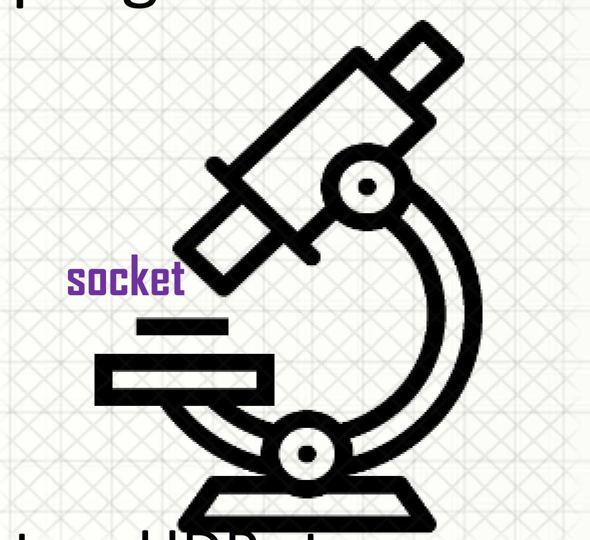
Use example, select()

- Assume the following scenario:
  - A client program sending a random message once in some random interval of time, repeatedly.
    - Each client has a unique ID.
  - A server program. The clients connect to the server, every time that the server receives a message, it prints “Client with ID X has sent a message”.
    - Several clients connect to the server at once.
- How can the server be implemented using select() ?

SOCKET PROGRAMING

Socket Programing

- In the first part, we will learn the fundamentals of socket programming.
- We will prepare 5 different short programs.
- Two for TCP.
  - A TCP receiver.
  - A TCP sender.
- Two for UDP
  - A UDP sender.
  - A UDP receiver capable of listening to a UDP stream.
- A program that can listen to serval UDP sockets at once.



Socket Programing

- Each sender program sends a number of messages with data from a file.
- Each receiver program receives this data and prints it.
- Each program will run on the lab's PCs, connected by a GNS3 topology from previous labs.
- We will use a new virtual machine with a GUI, PC5 – either write your programs in PC5 or use it as a middleman to a cloud.
  Connect it to a GNS3 topology and transfer your programs to the 4 lab PCs (via FTP or SCP)

Thanks to…(and other helpful links)

- Wikipedia
- http://www.cs.northwestern.edu/~agupta/cs340/sockets/sockets_intro.ppt
- http://parsys.eecs.uic.edu/~solworth/sockets.pdf
- https://www.cs.cmu.edu/~srini/15-441/S10/lectures/r01-sockets.pdf