Reti

Introduction to the lab activities

Programming tools

How to work in the lab activites

GCC

- command-line tool for compiling applications
- compile source code written in languages such as C and others into machine language so that it can be executed by the computer
- you will write and compile applications in C

➢ gcc server.c –o server

GDB

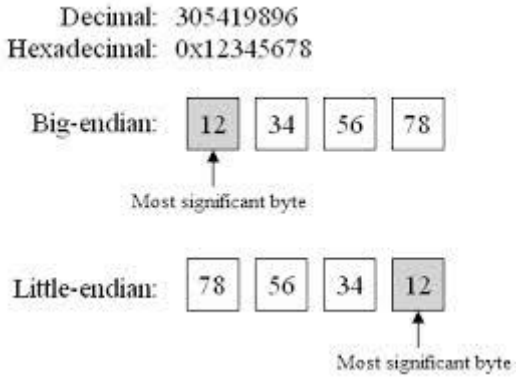
- command-line tool for debugging applications
- step-by-step execution
- print parameter values
- forcing set of parameter values
- GUI available with DDD

➢ gdb ./server

Byte order

Data stored in *Host Order*

The common order is the Network Order (aka Big-endian)



Byte order: conversion

Host to Network

before DATA goes out to the wire

Network to Host

convert DATA as they come in off the wire

Byte order: functions

htons():	**h**ost **to** **n**etwork **s**hort
htonl():	**h**ost **to** **n**etwork **l**ong
ntohs():	**n**etwork **to** **h**ost **s**hort
ntohl():	**n**etwork **to** **h**ost **l**ong

Useful tools

Troubleshooting without printf calls

What is my IP?

Ifconfig

lists all network interfaces of the machine

```
[user@host user]$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:E0:29:5E:FC:BE
          inet addr:192.168.2.1  Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:35772 errors:0 dropped:0 overruns:0 frame:0
          TX packets:24414 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:36335701 (34.6 Mb)  TX bytes:3089090 (2.9 Mb)
          Interrupt:5 Base address:0x6000
```

What is my IP?

The ‘ip’ command

- ip address show

```
[munafo@prezzemolo ~]$ ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s31f6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 2c:4d:54:cf:7b:ac brd ff:ff:ff:ff:ff:ff
    inet 130.192.9.131/24 brd 130.192.9.255 scope global noprefixroute enp0s31f6
       valid_lft forever preferred_lft forever
    inet6 fe80::541e:cc86:22f7:7fa6/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

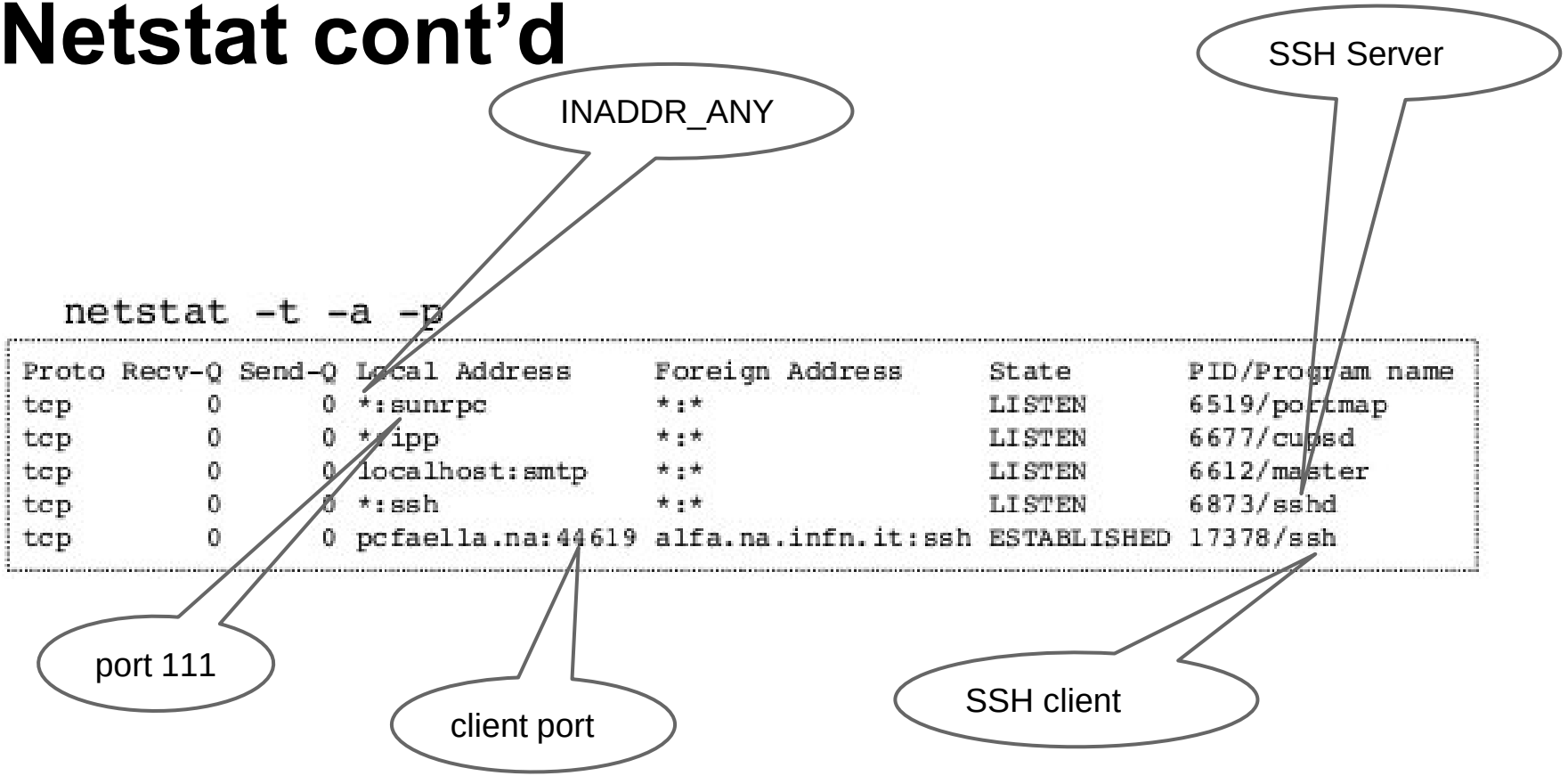
- ip link show

Netstat

list the socket activity on a particular machine

- -l : to see unconnected sockets (LISTEN state)
- -p : to see the process related to a socket
- -t : to see only TCP sockets
- -a : to see all listening and not-listening sockets

Netstat cont'd



lsof

Useful for discovering file descriptors associated with running processes (and thus sockets)

Preferable to use (also) the '-n' option to avoid address and name resolution

Netcat (or nc or ncat)

is able to send/receive any kind of data over TCP or UDP

very easy way to connect to a server and exercise it for debugging purposes

syntax: nc <host> <port>

server mode (listen): nc -l <host> <port>

SSH

Login into another machine

ssh <username>@<hostname>

Useful when for testing client/server running in different machines

Hints if your application is not working

- from client site, try to connect to the server
  - nc <serverIP> <serverPORT>
- from server site, check if the socket is listening
  - netstat -a | grep <serverPORT>
- make sure to compile the last version of your application
  - save the source code; make clean; make all

Socket programming

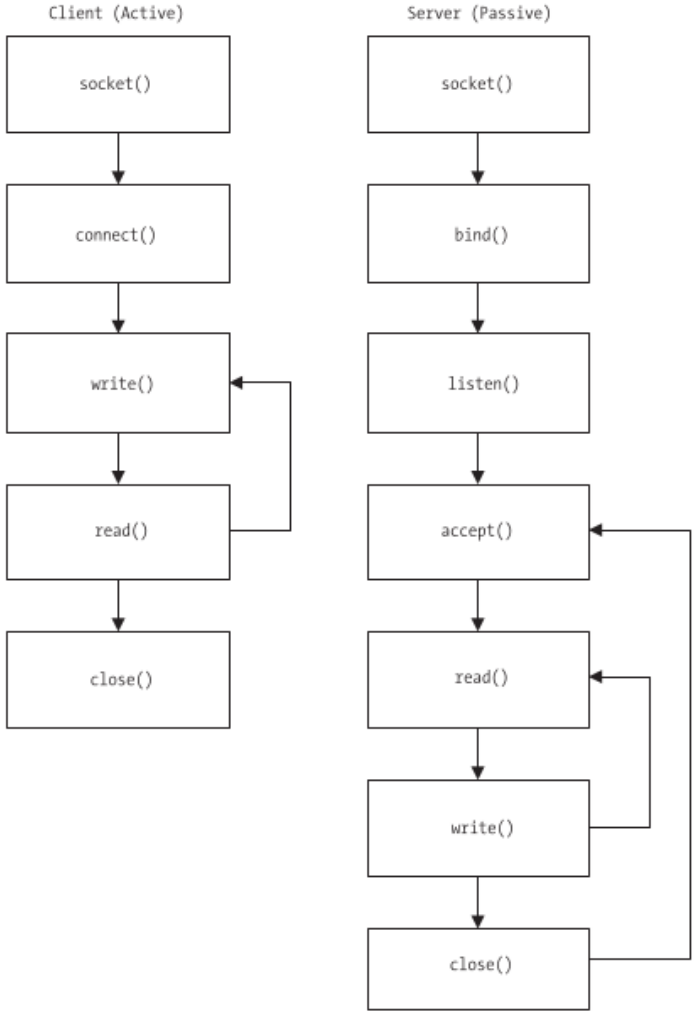
Berkeley socket interface

Socket

Abstraction for network communication

OPERATION	EXPLANATION
Open	Prepare for input or output operations.
Close	Stop previous operations and return resources.
Read	Get data and place in application memory.
Write	Put data from application memory and send control.
Control (ioctl)	Set options such as buffer sizes and connection behavior.

Function call flow



Socket constant

Define address family
Define type of service
Define protocol

Address family

ADDRESS FAMILY	DESCRIPTION
AF_UNIX, AF_LOCAL	Communications local to same host
AF_INET	IPv4 Internet protocols
AF_INET6	IPv6 Internet protocols
AF_IPX	IPX–Novell protocols
AF_NETLINK	Kernel user interface
AF_X25	X.25 protocols
AF_AX25	Amateur radio AX.25 protocols
AF_ATMPVC	ATM Private Virtual Circuits (PVCs)
AF_APPLETALK	AppleTalk protocols
AF_PACKET	Low-level packet communications

Socket type

TYPE	DESCRIPTION
SOCK_STREAM	Communications are connection-based, sequenced, reliable, and two-way.
SOCK_DGRAM	Connectionless, unreliable message-type communications using a fixed length.
SOCK_SEQPACKET	Message-type communications with fixed-length packets, but sequenced and more reliable.
SOCK_RAW	Access to raw network protocols.
SOCK_RDM	Connectionless but reliable communications, without using a particular packet order.
SOCK_PACKET	Obsolete and should not be used.

A call to socket()

```
mySocket = socket(AF_INET, SOCK_STREAM, 0);
```

Where endpoints are stored

```
struct sockaddr_in {
    short    sin_family;   /* type of address                 */
    u_short  sin_port;     /* protocol port number            */
                           /* network byte ordered            */
    u_long   sin_addr;     /* net address for the remote host */
                           /* network byte ordered            */
    char     sin_zero[8];  /* unused, set to zero             */
};
```

Socket programming

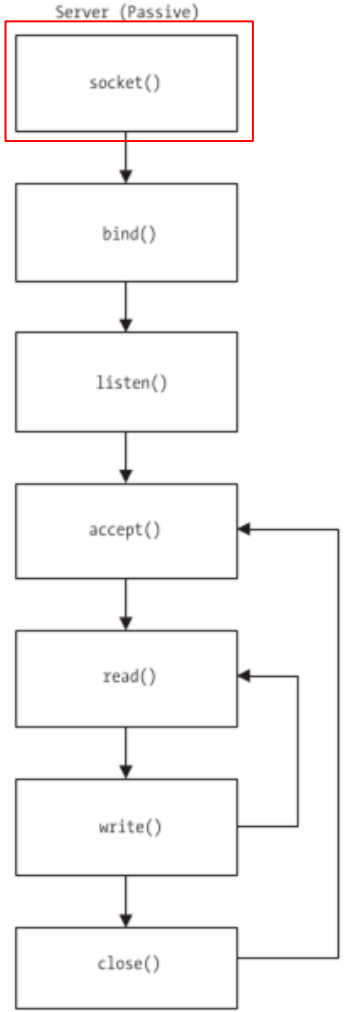
TCP Server and Client examples

Example: TCP Server

```
1.  #include <stdio.h>
2.  #include <sys/types.h>
3.  #include <sys/socket.h>
4.  #include <netdb.h>
5.  #include <stdlib.h>

6.  const char MESSAGE[] = "Hello UPO student!\n";

7.  int main(int argc, char *argv[]) {
8.          int simpleSocket = 0;
9.          int simplePort = 0;
10.         struct sockaddr_in simpleServer;

11. /* make sure we have a port number*/
12. if (2 != argc) {
13.          fprintf(stderr, "Usage: %s <port>\n", argv[0]);
14.          exit(1);
15. }
```

Example (II)

```
1. /* create a streaming socket*/
2. simpleSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

3. if (simpleSocket == -1) {
4.         fprintf(stderr, "Could not create a socket!\n");
5.         exit(1);
6. } else {
7.         fprintf(stderr, "Socket created!\n");
8. }
```



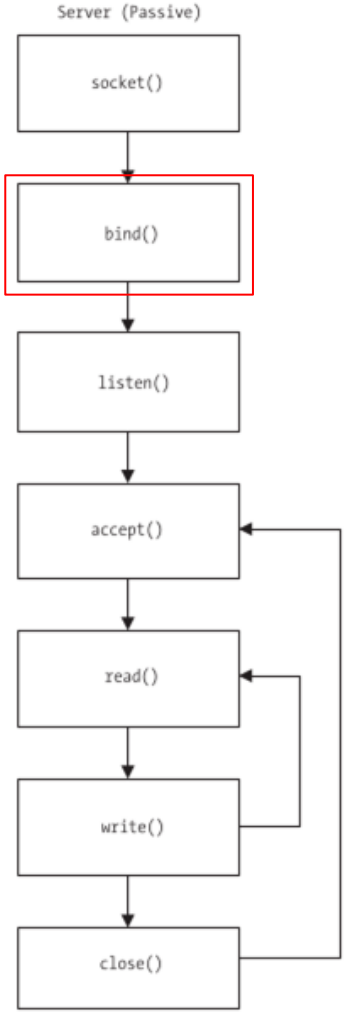
bind()

1. /* retrieve the port number for listening*/
2. simplePort = atoi(argv[1]);

3. /* set up the address structure */
4. /* use INADDR_ANY to bind to all local addresses */
5. /* note use of htonl() and htons() */
6. **memset(&simpleServer, '\0', sizeof(simpleServer));**
7. **simpleServer.sin_family = AF_INET;**
8. **simpleServer.sin_addr.s_addr = htonl(INADDR_ANY);**
9. **simpleServer.sin_port = htons(simplePort);**

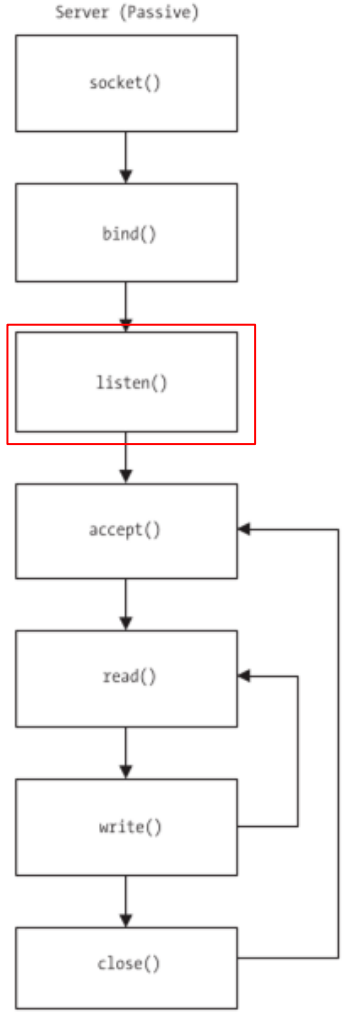
bind() -cont'd

```
1.  /* bind to the address and port with our socket */
2.  returnStatus = bind(simpleSocket,
3.                                  (struct sockaddr *)&simpleServer,
4.                                   sizeof(simpleServer));
5.  if (returnStatus == 0) {
6.                          fprintf(stderr, "Bind completed!\n");
7.  } else {
8.                          fprintf(stderr, "Could not bind to address!\n");
9.                          close(simpleSocket);
10.                         exit(1);
11. }
```

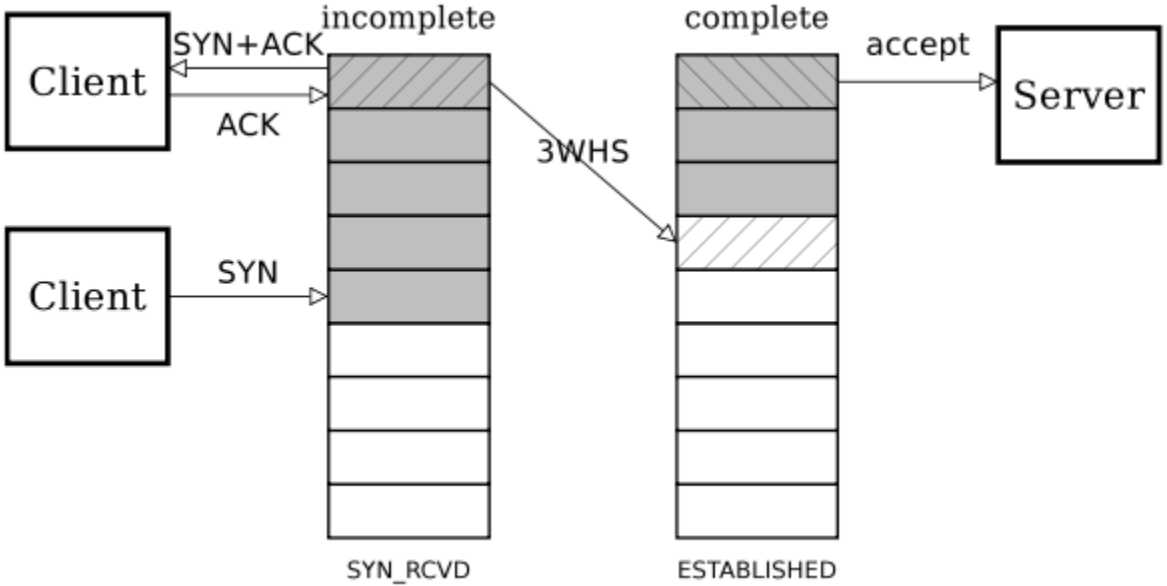


Listen

```
1.  /* lets listen on the socket for connections      */
2.  returnStatus = listen(simpleSocket, 5);

3.      if (returnStatus == -1) {
4.          fprintf(stderr, "Cannot listen on socket!\n");
5.          close(simpleSocket);
6.          exit(1);
7.      }
```

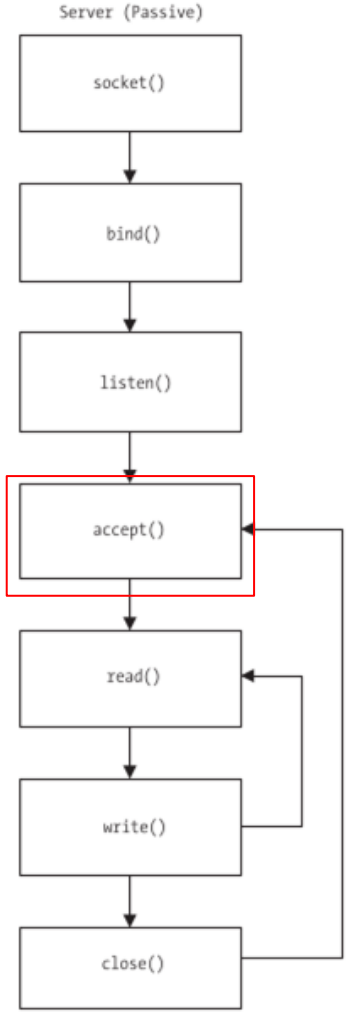


Listen cont’d



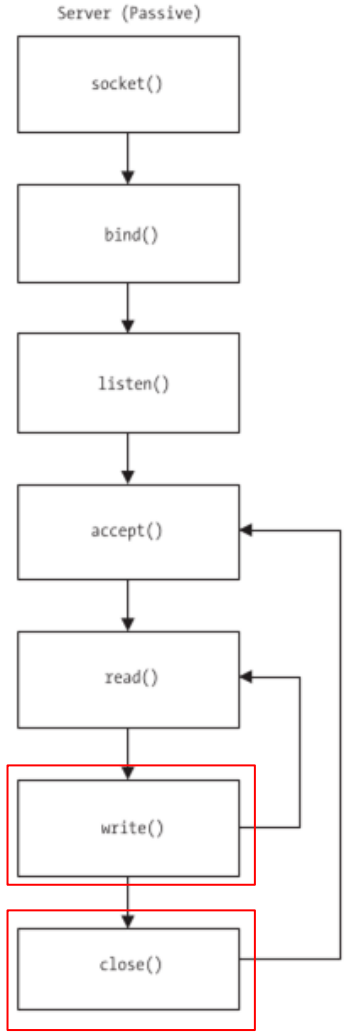
accept()

```
1.  while (1) {
2.          /* set up variables to handle client connections */
3.          struct sockaddr_in clientName = { 0 };
4.          int simpleClient = 0;
5.          int clientNameLength = sizeof(clientName);

6.          /* block on accept function call */
7.          simpleChildSocket = accept(simpleSocket,
8.                                     (struct sockaddr *)&clientName,
9.                                     &clientNameLength);
10.         if (simpleChildSocket == -1) {
11.                 fprintf(stderr, "Cannot accept connections!\n");
12.                 close(simpleSocket);
13.                 exit(1);
14.         }
```



write() and close()

```
1.                  /* handle the new connection request*/
2.                 /* write out our message to the client */
3.                 write(simpleChildSocket, MESSAGE, strlen(MESSAGE));
4.                 close(simpleChildSocket);
5.         } //end of while cycle
6.         close(simpleSocket);
7.         return 0;
8.   }
```

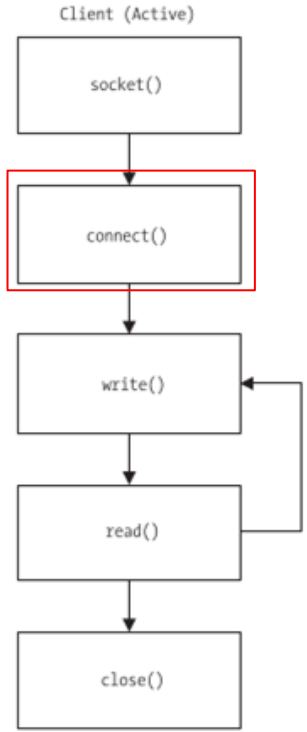


Example TCP client

1. simpleSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
2. ...
3. **simplePort = atoi(argv[2]);**

4. **memset(&simpleServer, '\0', sizeof(simpleServer));**
5. **simpleServer.sin_family = AF_INET;**
6. **simpleServer.sin_addr.s_addr=inet_addr(argv[1]);**
7. **simpleServer.sin_port = htons(simplePort);**
8. ... …
9. /* connect to the address and port with our socket*/
10. **returnStatus = connect(simpleSocket,**
    (struct sockaddr *)&simpleServer,
    sizeof(simpleServer));

    ...



Example TCP client (cont'd)

```
1. /* get the message from the server*/
2. returnStatus = read(simpleSocket, buffer, sizeof(buffer));

1. if ( returnStatus > 0 ) {
2.         printf("%d: %s", returnStatus, buffer);
3. } else {
4.         fprintf(stderr, "Return Status = %d \n", returnStatus);
5. }
```