Introduction to Computer Networks and the Internet COSC 264

Network Programming with Sockets

Slides prepared by [1]Dr. Andreas Willig
Updated and presented by [2]Dr. Barry Wu

[1]Department of Computer Science & Software Engineering
[2]Wireless Research Centre
College of Engineering

Semester 2, 2021

Scope

Goals:

- Explain the concept of sockets
- Introduce the socket API
- Show socket programming examples

Important

We discuss the “real thing”, i.e. the C socket interface. Other programming languages provide wrapper libraries. You will need to familiarize yourself with the relevant Python functions, there are plenty of programming examples in the Internet.

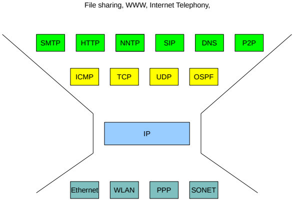
Important

You can use the socket interface without understanding the operation of the underlying protocols (TCP, UDP, IP). What matters are the services offered by these protocols!

Outline

Outline

The Role of IP in the Internet Protocol Stack



- IP = Internet Protocol
- There are two protocol versions (4, 6), here we use IPv4
- “Everything over IP, IP over everything”

IP Addresses

- Each host is identified by one or more **IP addresses**
  - A host has as many IP addresses as it has network adapters
  - End hosts usually only have one IP address
- The IP address not only identifies the host, but also helps the network to find a path to this host
- Humans normally do not work with IP addresses directly but with human-readable host names like `www.canterbury.ac.nz`
- There exists a special service / protocol called the *domain name service* (DNS) which translates human-readable host names to IPv4 addresses, the Internet itself only works with IPv4 addresses

IP Address Representation

- IPv4 addresses have a width of 32 bits
- They are supposed to be worldwide unique
- IP addresses are written in **dotted-decimal notation**, e.g.:

  `130.149.49.77`

  where decimal numbers are separated by dots

IP Service – Best Effort

- Basic IP service is **packet delivery**
- This service is:
  - Connectionless: no connection or shared state is set up before datagram delivery starts
  - Unacknowledged: IP does not use acknowledgements
  - Unreliable: on IP level no retransmissions are carried out
  - Unordered: IP does not guarantee in-sequence delivery [1]
- It is very similar to sending postcards
- This kind of guarantee-nothing service is called **best effort**

Outline

UDP and TCP

- In the Internet there are two further foundational protocols:
  - UDP – User Datagram Protocol
  - TCP – Transmission Control Protocol
- Both protocols operate *on top* of IPv4, they generate packets and stuff these as payload data into IPv4 packets
  - This is related to **protocol layering**
- UDP or TCP sessions run between a pair of IP addresses
- They add own addressing capabilities (*port numbers*). Analogy:
  - An IPv4 address corresponds to the street address of a house
  - A port number refers to an individual person in that house
- In a networking context:
  - An IPv4 address refers to an end host
  - A port number refers to a particular application / server software running in that end host (out of possibly several)
- Through these port numbers, there can be many simultaneous “conversations” going on between the same two hosts

UDP

- UDP = User Datagram Protocol
- UDP offers a service that is similar to the service of IPv4:
  - Connectionless
  - Unacknowledged
  - Unreliable
  - Unordered
- Almost the only thing that UDP really adds are port numbers

TCP

- TCP = Transmission Control Protocol
- The TCP service is:
  - Connection-oriented
  - Reliable and in-sequence
  - Byte-stream oriented
  - Full-duplex
- In other words: using the TCP protocol allows for “safe and reliable” data transfer over the Internet, where a string of bytes is transferred reliably and in the right sequence
- To do this, TCP does a lot of work under the hood . . .

Ports and Application Multiplexing

- IP addressing allows to distinguish network interfaces
- To address different applications / processes / tasks / threads in a host, an additional addressing layer is needed
- UDP and TCP accomplish this by **ports**:
  - A port is identified by a 16-bit number (**port number**)
  - A process on a host can allocate one or more ports
  - One port is allocated to at most one application / process
- Some ports are often allocated to well-known applications / higher-layer protocols (**well-known ports**), e.g.:
  - Port 80: HTTP (World Wide Web, runs over TCP)
  - Port 22: SSH (Secure Shell, runs over TCP)
  - Port 25: SMTP (Email, runs over TCP)
  - Port 53: DNS (runs over UDP or TCP)
- **But**: this is pure convention, not enforced!

Outline

Processes and Blocking Socket Calls

- Some socket calls (e.g. `accept() or select()`) have an important property: they can be **blocking**
- This refers to the concept of processes in operating systems (OS) like Windows / Mac OS X / Linux
- A process is a running instance of a program / application
- Operating systems create an abstraction for processes that:
  - makes them believe they have their own exclusive processor
  - allocates memory exclusively (not accessible by other processes)
  - are the units that can request other OS resources like sockets, pipes, files, etc

Processes and Blocking Socket Calls (2)

- In reality the OS uses a time-sharing approach to let several processes run on one processor
- Processes can be in one of several states:
  - Running: actually running on the processor
  - Runnable: ready to run on a processor, but currently not doing so
  - Blocked: process is waiting for input from a hard disk / socket / . . .
    – when input arrives the process becomes runnable or running
    – while blocked, the process does not use any CPU cycles
- A call like `select()`:
  - Waits for input on one or more sockets (e.g. incoming packets)
  - If no input is available at time of call, the calling process will be put into blocked state, hence does not use the CPU
  - This is better than the alternative called **busy-looping**, in which a process repeatedly checks the sockets in a loop to see whether new data has arrived – this consumes a lot of CPU resources

Outline

Introduction

- The socket API has been introduced with the 4.2BSD Unix operating system in 1983
- It follows the Unix philosophy that (almost) everything is a file or could be made to look like a file, programmers can use sockets in a way similar to using files
- All Internet applications like web, mail, FTP etc. work with sockets

Ports and Sockets

- Applications use the programming abstraction of a **socket**
  - A socket is bound to exactly one port
    - Several sockets can be bound to same port (within the same process)
  - A socket is associated with underlying protocol (often UDP or TCP)
    - Note: the application does not need to know how or when the protocol operates precisely, but it needs to know the *service*!
  - A socket is associated with **buffers**:
    - These buffers decouple the application from the underlying protocol
    - Receiving UDP / TCP protocol entity places incoming data into receive buffer, applications `read()` data from buffer at their leisure
    - Application at transmitter `write()`'s data into transmit buffer, TCP/UDP entity sends it at its discretion
  - It is possible to both transmit and receive from a socket
- Conceptually, a socket has similarities with file handles

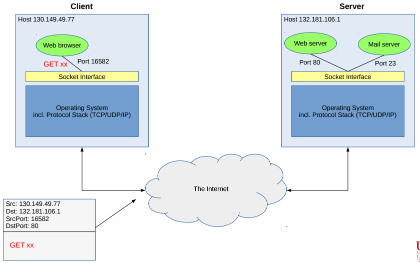
Ports and Sockets (2)

- The socket API has some overlap to the Unix file API:
  - Sending over a socket can be achieved by calling `write()`
  - For reading from a socket you can call `read()`
  - Sockets can be used the same way as filehandles e.g. in `select()`
- When writing to a socket:
  - UDP: the data is encapsulated in UDP datagram and transferred – each `write()` leads to separate datagram
  - TCP: the data is buffered and possibly combined with data from further `write()`'s before transmitting
- **Careful**: a successful `write()` call does not mean that the data has already been successfully received, only that the data could be placed in the sender's socket buffer!

Client/Server Paradigm in the UDP/TCP/IP Context

- Client applications:
  - run on an end host
  - A client needs to know servers IP address / port number, needs to have an own IP address / port number and a socket bound to these
  - It then initiates contact with the server "through its socket"
  - Afterwards a client application can end
- Server applications:
  - run on an end host
  - need an IP address / port number that is known to clients and need to have a socket bound to these
  - accept service requests from client and respond to them
  - usually run all the time and support several clients
- Client and server applications can even run on the same machine and communicate through the socket interface
  - Client uses special IP address `127.0.0.1`, known as **localhost**

Client/Server Paradigm in the UDP/TCP/IP Context (2)



Socket Types

- When a socket is created, it can be of one of several types
- A **stream socket**:
  - Offers reliable, in-sequence delivery of a byte-stream
  - Is built on the TCP protocol
  - If delivery fails for some reason, the sending application is notified
- A **datagram socket**:
  - Does not guarantee reliable or in-sequence delivery
  - Is built on the UDP protocol
  - If delivery fails the sending application is not informed
- These are the main types, there are other types:
  - **Raw sockets**: bypass UDP or TCP and directly send IP packets
  - **Sequenced-packet sockets**: a variant of a stream socket
  - Further types might be available in some operating systems

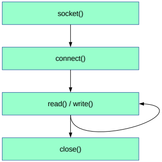
Socket Types (2)

- Datagram sockets offer a datagram service
  - when `write()`ing x bytes to a datagram socket the written data is encapsulated into a UDP packet and sent out
  - the receiver will `read()` a block of at most x bytes from its socket
- Stream sockets offer a **byte-stream service**:
  - Smallest unit of transmission is one byte
  - Different `write()`s do not necessarily lead to different packets
  - Receiver cannot easily detect which was last byte supplied at the transmitter with one `write()` command, e.g.:
    - Sender performs `write()` with ten bytes, followed by `write()` with ten bytes, followed by `write()` with 20 bytes
    - Receiver `read()`s one piece of 40 bytes and cannot tell how many `write()`s contributed to it, nor where boundaries were
- In other words: datagram sockets preserve record boundaries, stream sockets do not
- Sequenced-packet sockets preserve record boundaries

Outline

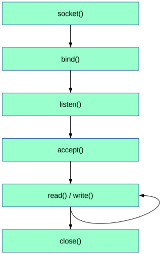
Outline

Example Workflow: TCP Client



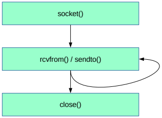
- `socket()`: create a socket, this allocates resources (e.g. buffer, socket handle), and assigns a **random unused port number**
- `connect()`: establish connection with server
- `read()`/`write()`: reading / writing from / to socket. Alternatively one can also use `sendto()` and `recvfrom()`, which allow to specify additional parameters
- `close()`: close connection

Example Workflow: TCP Server



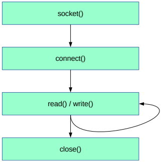
- `bind()`: bind a socket to a particular port number and IP address
- `listen()`: declare willingness to accept incoming connections, allocate resources (queue) for incoming connection requests (TCP)
- `accept()`: accept an incoming connection request (i.e. take it from the connection request queue) and create new socket for this connection, bound to the same port as the socket we called `listen()` on

Example Workflow: UDP Client



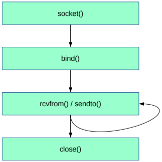
- Compared to TCP client no `connect()` system call is necessary
- Receiver address then has to be supplied to `sendto()`
- `recvfrom()` receives data and provides address of transmitter

Example Workflow: UDP Client with `connect()`



- The `connect()` system call supplies a default receiver address
- `connect()` can be called multiple times to change the default receiver
- `write()` then sends to the last receiver specified with `connect()`

Example Workflow: UDP Server



- Compared to TCP server no `listen()`, and `accept()` calls are necessary

Outline

Main Socket API Functions

- We only discuss the most important socket calls
- We use the Linux socket API as a guideline, but we will not discuss all available options
- For each function you find a `man` page in `man` section 2, e.g.:

```
man 2 socket
```

- We will also indicate the C include files that are needed
- All socket calls set the `errno` variable to indicate the cause of an error, the errors that can be raised by a particular socket call are described in its `man` page

`socket()`

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

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

- Creates a new socket structure, including allocation of resources like the socket buffer, assigns a random un-used port number to it, and returns a file descriptor representing the socket
- `socket()` is non-blocking
- Parameter `domain`:
  - Selects the protocol family to be used for the socket
  - Options include: `AF_INET` for IPv4, `AF_INET6` for IPv6, `AF_APPLETALK` for the Appletalk protocol

`socket()` (2)

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

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

- Parameter `type`:
  - Indicates socket type
  - Options include `SOCK_STREAM` for a stream socket, `SOCK_DGRAM` for a datagram socket, `SOCK_RAW` for a raw socket
- Parameter `protocol`:
  - Selects the protocol used for the given socket type
  - Often only one option sensible, then `protocol=0` is a good choice
- Return value:
  - If successful, a file descriptor (≥ 0) is returned
  - Otherwise, -1 is returned and error code `errno` is set

`bind()`

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- Links a socket to a particular IP address / port number / address family combination
- `bind()` is non-blocking
- Parameter `sockfd`:
  - Denotes the socket
  - Is just the value returned by previous `socket()` call
- Return value:
  - When operation successful: 0
  - Otherwise: -1, and error is indicated in variable `errno`

bind() – The sockaddr Structure

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

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

struct sockaddr_in {
    short            sin_family;
    unsigned short   sin_port;
    struct in_addr   sin_addr;
    char             sin_zero[8];
};

struct in_addr {
    unsigned long s_addr;
};
```


bind() – The sockaddr Structure (2)

- The socket API is meant to support a whole range of underlying networking protocol stacks, not only IPv4, but also IPv6, X25, ATM and others, and all of these have different address representations
- The `domain` parameter of `socket()` indicates which protocol stack and address family (`AF_x`) you want to use
- For the IPv4 domain one works with a `sockaddr_in` struct and typecasts this to a `sockaddr` pointer when calling `bind()`
- See the examples below

listen()

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

int listen(int sockfd, int backlog);
```

- `listen()` declares your willingness to accept incoming stream connections (TCP) of type `SOCK_STREAM` or `SOCK_SEQPACKET`, allocates resources like a queue for incoming connection requests
- It is only used on the server side
- Parameter `sockfd`:
  - Denotes the socket
  - Is just the value returned by previous `socket()` call

listen() (2)

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

int listen(int sockfd, int backlog);
```

- Parameter `backlog`:
  - There is a queue for yet un-processed incoming connection requests and `backlog` specifies how long this queue can be
  - Any excess connection request is declined (i.e. a packet is sent back to the client informing it of denial)
- Return value:
  - If successful: 0
  - Otherwise: -1 and `errno` is set
- `listen()` is non-blocking

accept()

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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- `accept()` waits for incoming connection requests of stream type (TCP), i.e. of type `SOCK_STREAM` or `SOCK_SEQPACKET`
- For an incoming connection a new socket is created (with the same port number as the socket on which previously `listen()` was called) and its descriptor returned as return value
- This is only used on a stream (TCP) server
- Parameter `sockfd`:
  - Denotes the socket on which you accept connections
  - You must have called `listen()` on this socket before

accept() (2)

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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- Parameters `addr` and `addrlen`:
  - Here you can receive address information about the requesting client (e.g. its IP address, port number and address family)
  - The caller has to allocate a buffer and pass its address and length to `accept()`
  - The precise type of the address information depends on the address family, for IPv4 you can typecast that into type `sockaddr_in`
- `accept()` is blocking (you can change that through options)

connect()

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

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- For stream (TCP) sockets (type `SOCK_STREAM` or `SOCK_SEQPACKET`) a client uses `connect()` to request establishment of a connection (e.g. TCP connection) with a server
- For datagram (UDP) sockets (type `SOCK_DGRAM`) a client uses `connect()` to specify a default receiver for datagrams that is used when `send()` or `write()` are used for data transmission
- Parameter `sockfd`:
  - Denotes the socket over which you want to communicate
  - Is just the value returned by `socket()`

connect() (2)

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

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- Parameters `addr` and `addrlen`
  - You have to prepare an address structure containing the address / port number / address family of the server or intended receiver
  - Format of this depends on address family
  - For IPv4 a `sockaddr_in` structure can be used, which can be type-cast to a `sockaddr` struct
- Return value:
  - On success: 0
  - Otherwise: -1 and `errno` is set
- `connect()` is blocking (until fate of connection setup is known)

read() and Friends

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

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
ssize_t read(int fd, void *buf, size_t count);
```

- These functions can read received data from a socket
- Caller provides a buffer `buf` of given length `len` into which received data will be written
- In `recvfrom()` the caller also provides memory for an address structure (e.g. of type `sockaddr_in`) in which the receiving protocol provides the address / port / address family of the host from which data was received

read() and Friends (2)

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

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
ssize_t read(int fd, void *buf, size_t count);
```

- `read()` is used for reading from a file indicated by the `fd` argument (file descriptor), here you can supply the socket descriptor `sockfd` you got from `socket()`
- Return value:
  - If it is ≥ 0 then it denotes the *actual* number of bytes read (which can differ from the requested number of bytes to read)
  - If it is -1 then an error occured, reason noted in `errno`
- All these functions are normally blocking, but can be given options (in the `flag` parameter) to make them non-blocking

write() and Friends

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

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t write(int fd, const void *buf, size_t count);
```

- For all these functions the data that is to be sent is supplied in the `buf` and `len` parameters, pointing to a buffer
- `send()` and `write()` require that the addressee has been specified before by `connect()`

write() and Friends (2)

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

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t write(int fd, const void *buf, size_t count);
```

- With `sendto()` the addressee has to be explicitly specified in the `dest_addr` and `addrlen` parameters, which in case of the IPv4 address family can be of type `sockaddr_in`
- Return value:
  - If it is ≥ 0 then it denotes the actual number of bytes written
  - If it is -1 then an error occured and `errno` contains information

`write()` and Friends (3)

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

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t write(int fd, const void *buf, size_t count);
```

- When the underlying socket buffer has still enough space to accommodate data, these functions are non-blocking, but if there is not enough space, the caller is blocked
- An important note about the semantics:
  - When these functions return “successfully” you do not know more than that the data has been placed into the socket buffer
  - You cannot make any inference on whether or not the data has actually been transmitted or successfully received

`close()`

```
#include <unistd.h>

int close(int fd);
```

- Needs to be called whenever you do not need a socket anymore
- Reason: it gives back the socket resources (buffer, file descriptor) to the operating system for use by other applications
- As argument you supply the socket descriptor `sockfd` returned by `socket`

Other Important Functions

- `poll()`
- `select()`
- You find out what these do . . .

Outline

Memory Representation and Endianness

- Processors can use different ways to store data in memory
- This applies in particular to values that are relevant to Internet protocols, such as 16-bit port numbers or 32-bit IPv4 addresses
- Endianness is the byte order in data of multiple bytes in memory

A 16-Bit Example

- We are given a 16-bit integer number

$$b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$$

- Such a number must be stored somehow in memory, which we imagine as an ordered sequence of bytes
- Intel 80x86 processors use the *low-endian* format

$$b_7b_6b_5b_4b_3b_2b_1b_0 \qquad b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8$$

  where the less significant byte sits at lower address and the more significant byte sits at higher address.
- Motorola processors (e.g. 68000 family) or PowerPC processors use *big-endian*

$$b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8 \qquad b_7b_6b_5b_4b_3b_2b_1b_0$$

- Now imagine that a sender with Intel processor sends a port number to a Motorola receiver

Memory Representation and Endianness (2)

- Therefore, for certain data types relevant for networking purposes (e.g. unsigned 16-bit integers, unsigned 32-bit integers), and in particular fields in a packet header, one:
  - standardizes / defines a *canonical representation* that is actually being transmitted – called the *network byte order*
  - asks hosts to convert between their own internal representation (*host byte order*) and network representation as needed
- In the Internet the canonical representation is big-endian

Helper Functions for Conversion

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

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

- These functions convert from host(h) to network(n) representation or vice versa
- They exist for 16 bit (short) and 32 bit (long) numbers
- These helper functions are described in `man` section 3

Helper Functions for Address Manipulation

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

int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
in_addr_t inet_network(const char *cp);
char *inet_ntoa(struct in_addr in);
struct in_addr inet_makeaddr(int net, int host);
in_addr_t inet_lnaof(struct in_addr in);
in_addr_t inet_netof(struct in_addr in);
```

- It is often useful to convert from string representations of addresses like "130.149.49.77" to 32-bit address values
- The above helpers support this type of conversion
- These helper functions are described in `man` section 3

Outline

Socket Programming Examples

- We will go through two examples:
  - A TCP server
  - A TCP client
- The code has been taken from `http://www.linuxhowtos.org/C_C++/socket.htm`, where you find the source code discussed here

Outline

TCP Client – (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(const char *msg)
{
    perror(msg);
    exit(0);
}
```

- `#include` statements make library functions known
- `perror()` prints given message on `stderr`
- `exit()` stops a program, returns given exit code

TCP Client – (2)

```
int main(int argc, char *argv[]) // argc=no of arguments, argv=list of arg
{
  int sockfd, portno, n;
  struct sockaddr_in serv_addr; // address structure for the server
  struct hostent *server;       // result of DNS resolver

  char buffer[256];             // buffer for actual data
  if (argc < 3) {               // check number command line arguments
     fprintf(stderr,"usage %s hostname port\n", argv[0]);
     exit(0);
  }
  portno = atoi(argv[2]);       // convert port number argument to intege

  // now create socket, protocol=0 means to use default protocol
  // for given address family (here: TCP will be used)
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
      error("ERROR opening socket");
```


TCP Client – (3)

```
// get server name (as text) from command line, and convert it
// to an IP address by using the DNS resolver library
// The resulting IP address is a 32 bit number in network byte
// order stored in the component server->h_addr
server = gethostbyname(argv[1]);
if (server == NULL) {
    fprintf(stderr,"ERROR, no such host\n");
    exit(0);
}
```

- Note: `gethostbyname()` is deprecated now under Linux, instead you would use `getaddrinfo()`

TCP Client – (4)

```
// now set up the sockaddr_in structure for the connect call
// zero out memory
bzero((char *) &serv_addr, sizeof(serv_addr));
// set address family
serv_addr.sin_family = AF_INET;
// copy server address from DNS result structure
bcopy((char *)server->h_addr,
     (char *)&serv_addr.sin_addr.s_addr,
     server->h_length);
// set port number in network byte order
serv_addr.sin_port = htons(portno);
// Finally the call to connect()
if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr))<0)
    error("ERROR connecting");
```

TCP Client – (5)

```
// get some data to send from the user and store it
// in buffer
printf("Please enter the message: ");
bzero(buffer,256);
fgets(buffer,255,stdin);

// send the data
n = write(sockfd,buffer,strlen(buffer));
if (n < 0)
     error("ERROR writing to socket");

// prepare buffer for receiving response data, wait
// for response data and print it
bzero(buffer,256);
n = read(sockfd,buffer,255);
if (n < 0)
     error("ERROR reading from socket");
printf("%s\n",buffer);
```


TCP Client – (6)

```
  // Clean up and exit
  close(sockfd);
  return 0;
}
```


Outline

TCP Server – (1)

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

void error(const char *msg)
{
    perror(msg);
    exit(1);
}
```

TCP Server – (2)

```
int main(int argc, char *argv[])
{
   int sockfd, newsockfd, portno;
   socklen_t clilen;
   char buffer[256];
   struct sockaddr_in serv_addr, cli_addr;
   int n;

   // check number of command line arguments, port number needed
   if (argc < 2) {
       fprintf(stderr,"ERROR, no port provided\n");
       exit(1);
   }

   // create the socket
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0)
      error("ERROR opening socket");
```


TCP Server – (3)

```
   // bind the socket to the port number given in the
   // command line, on which the server shall receive
   // incoming requests
   // Using an address of INADDR_ANY means that the socket
   // will be bound to *all* local interfaces (there can be
   // more than one). If only one specific interface is to
   // be used, then the IP address of that interface needs
   // to be given
   bzero((char *) &serv_addr, sizeof(serv_addr));
   portno = atoi(argv[1]);
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = INADDR_ANY;
   serv_addr.sin_port = htons(portno);
   if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))<0)
            error("ERROR on binding");
```


TCP Server – (4)

```
   // declare that we will listen on this socket for incoming
   // requests, allow maximum of five queued connection
   // requests
   listen(sockfd,5);


   // Wait (blocking) for incoming connection requests.
   // The client address information will be provided in
   // cli_addr
   // we get a new socket newsockfd over which we will handle
   // the client request
   clilen = sizeof(cli_addr);
   newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
   if (newsockfd < 0)
        error("ERROR on accept");
```


TCP Server – (5)

```
   // read data from newsockfd into buffer and respond to it
   bzero(buffer,256);
   n = read(newsockfd,buffer,255);
   if (n < 0) error("ERROR reading from socket");
   printf("Here is the message: %s\n",buffer);
   n = write(newsockfd,"I got your message",18);
   if (n < 0) error("ERROR writing to socket");

   // clean up and exit
   close(newsockfd);
   close(sockfd);
   return 0;
}
```

[1] Jon C. R. Bennett, Craig Partridge, and Nicholas Shectman.
Packet Reordering is Not Pathological Network Behaviour.
IEEE/ACM Transactions on Networking, 7(6):789–798, December 1999.