

CSC2/452 Computer Organization Networking

Sreepathi Pai

URCS

November 28, 2022

Outline

Administrivia

Networking

The Internet Protocol (IP)

The Sockets Client Programming Interface

The Sockets Server Programming Interface

An Echo Server and Client

Outline

Administrivia

Networking

The Internet Protocol (IP)

The Sockets Client Programming Interface

The Sockets Server Programming Interface

An Echo Server and Client

Administrivia

- ▶ Assignment #2 Parts 1 and Part 2 Redux will be out tonight at 7PM
 - ▶ Due date: Wednesday, Nov 30, 7PM
 - ▶ No late submissions allowed
- ▶ Assignment #5 (last!) will be out Dec 2

Outline

Administrivia

Networking

The Internet Protocol (IP)

The Sockets Client Programming Interface

The Sockets Server Programming Interface

An Echo Server and Client

Communication between computers

- ▶ Getting different computers to work together requires networking
- ▶ Many different ways of physically building a network
 - ▶ Physical: Copper, Optical fibre, Wireless, etc.
 - ▶ Technologies: Ethernet, InfiniBand, GSM, CDMA, 802.11 etc.
- ▶ Lots of physical (and logical) components
 - ▶ Nodes, bridges, routers, etc.
- ▶ Ultimately, though, software needs to use the network
 - ▶ The focus of today's class
 - ▶ See the textbook for details on network organization
 - ▶ Or take CSC2/457

Streams and Packets

- ▶ Stream-oriented communication
 - ▶ Data is sent in some order and must be received in that same order
 - ▶ Usually because data bytes are ordered (e.g. contents of a file)
 - ▶ Communication must be reliable, data bytes must not be lost and errors must be detected
 - ▶ Usually long-lived
- ▶ Packet-oriented communication
 - ▶ Data is sent as “packets”, independent pieces of data
 - ▶ Packets are not ordered, can be delivered out-of-order
 - ▶ Packets may be lost, corrupted or duplicated
 - ▶ Communication usually does not last longer than a single packet

Circuit-switching vs Packet switching

- ▶ Switching: determining the path taken by data
- ▶ Circuit-switching
 - ▶ Establish a circuit from source to destination once at beginning of connection
 - ▶ Data takes the path established by the circuit
 - ▶ Beloved of telephone companies
 - ▶ e.g. ISDN, ATM (not the machines dispensing money)
- ▶ Packet-switching
 - ▶ No fixed path from source to destination
 - ▶ Packets choose their own path, on a hop-by-hop basis
 - ▶ Foundation of the Internet
 - ▶ e.g. Internet Protocol

Stream-oriented communication using packets?

- ▶ ARPA chose packet-switching for the initial design of the Internet because it was highly resilient
 - ▶ Imagine the network as a graph, with computers as nodes and communication links as edges
 - ▶ Packet-switching can get the data across even when many links and nodes fail
 - ▶ Usually by choosing a path around these failures
- ▶ But, packets can:
 - ▶ be reordered
 - ▶ can be corrupted
 - ▶ can get lost
- ▶ How to build an abstraction of stream-oriented communication over a packet-switched network?

Outline

Administrivia

Networking

The Internet Protocol (IP)

The Sockets Client Programming Interface

The Sockets Server Programming Interface

An Echo Server and Client

Internet Protocol

- ▶ The building block of the Internet
 - ▶ two major versions: IPv4 and IPv6
- ▶ IP is designed to connect networks (hence *internet*(working))
 - ▶ Typically, machines on a single network can talk directly to each other
 - ▶ IP specifies how to route packets across networks
- ▶ All nodes in the network have an address
 - ▶ IPv4: 32-bit addresses
 - ▶ IPv6: 128-bit addresses
- ▶ IP usually runs over a local network protocol
 - ▶ Usually, Ethernet for wired networks

Ethernet: A local network protocol

- ▶ Designed at Xerox PARC by Bob Metcalfe
- ▶ Originally designed for a shared broadcast medium
 - ▶ All computers share the same communication medium
 - ▶ Two (or more) computers cannot talk at the same time
 - ▶ Key question: how to allow computers to share this medium?

How Ethernet solves the shared medium problem

- ▶ Computers check if a communication is in progress
- ▶ If not, they start communicating
- ▶ If two computers started communicating at the same time, they'll interfere with each other
 - ▶ Called a *collision*
 - ▶ These collisions can be detected
- ▶ Both computers will stop and try again after waiting a *random* interval of time
 - ▶ This interval of time will increase exponentially on each unsuccessful attempt
 - ▶ Called *random (or exponential) backoff*
- ▶ Alternatives to Ethernet: Token Ring

How IP works

```
$ route -n
Kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0          192.168.1.1    0.0.0.0         UG    600    0      0 wlan0
192.168.1.0      0.0.0.0        255.255.255.0   U     600    0      0 wlan0
```

- ▶ Each node contains a routing table (you can use the command `route`) to print it out
- ▶ The routing table contains information on how to route packets
- ▶ In the example above
 - ▶ addresses 192.168.1.0 to 192.168.1.255 are local and packets should be sent directly via interface `wlan0`
 - ▶ all other addresses should be sent to the gateway at 192.168.1.1 which will forward them
- ▶ These are IPv4 addresses, consisting of four 8-bit numbers separated by periods.

Local IP transmission

- ▶ If 192.168.1.5 wants to send packets to 192.168.1.1, it can send them directly
- ▶ However, this requires knowing the MAC address of 192.168.1.1
 - ▶ Media Access Control, i.e. the “Ethernet” address
- ▶ Usually, this is done by broadcasting a query to all local nodes: “who is 192.168.1.1?”
 - ▶ This is called the address resolution protocol (or ARP)
 - ▶ The result is cached to avoid frequent broadcasts
- ▶ Once a MAC address is obtained, Ethernet can be used to directly send the packet

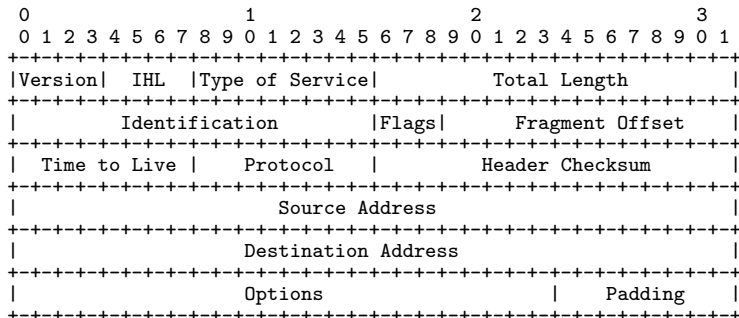
Non-local IP transmission

- ▶ If 192.168.1.5 wants to send packets to 8.8.8.8, it cannot send them directly
- ▶ For non-local addresses, packets are sent to the gateway
 - ▶ The gateway is on the local network
- ▶ The gateway is connected to at least two networks
 - ▶ If the packet is intended for one of its two networks, it can do local delivery
 - ▶ Otherwise, it forwards the packet to the next gateway
- ▶ The forwarding step can be made faster using “routers”
 - ▶ Gateways that discover and remember the fastest path to a network

IP Data transmission algorithm

- ▶ If address is local
 - ▶ find MAC address using ARP (if not known)
 - ▶ send packet directly using link-layer protocol (i.e. Ethernet)
- ▶ If address is non-local
 - ▶ send packet to gateway

IP Packets



- ▶ TTL: field is decremented by one by each gateway/router
 - ▶ Packet discarded when the field reaches 0
- ▶ Protocol: identifies next-level protocol
 - ▶ Ethernet at bottom, followed by IP, followed by ...

(from historical standard RFC 791)

Example IP Packet (or 'Datagram')

```

0                               1                               2                               3
0 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|Ver= 4 |IHL= 5 |Type of Service|                Total Length = 21    |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Identification = 111      |Flg=0|  Fragment Offset = 0    |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  Time = 123  |  Protocol = 1  |                header checksum    |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     source address                |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     destination address            |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      data      |
+-----+-----+-----+-----+-----+-----+-----+-----+

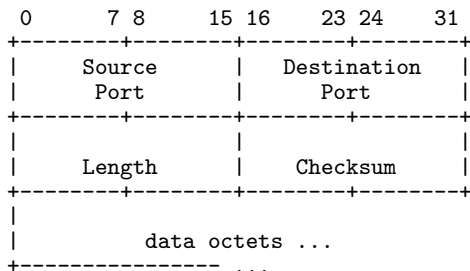
```

- ▶ Protocol 1 is Internet Control Message Protocol (ICMP)
 - ▶ The protocol used for sending pings
 - ▶ Or network error messages ("Destination unreachable")
 - ▶ BTW, this is not a valid ICMP packet, just an example

TCP and UDP

- ▶ Like ICMP, these are next level protocols built on top of IP
- ▶ User Datagram Protocol or UDP
 - ▶ Provides user-level (think application-level) packet communication
 - ▶ Recall, IP is for node-to-node communication
- ▶ Transmission Control Protocol or TCP
 - ▶ Provides a reliable, stream-oriented communication on top of IP
 - ▶ Also user/application-level

UDP



- ▶ UDP extends IP addresses with the notion of a *port*
 - ▶ A port is a 16-bit unsigned number (0 to 65535)
- ▶ An application “listens” on a well-known port
 - ▶ e.g. 8.8.8.8:53, where 53 is the port on node 8.8.8.8
 - ▶ data addressed to a port is forwarded to the application
- ▶ UDP packets are encapsulated in IP
 - ▶ I.e. a UDP packet is the “data” portion of an IP packet
 - ▶ Protocol is set to 17

TCP

- ▶ TCP must:
 - ▶ provided stream-like data transfer
 - ▶ deliver data in order
 - ▶ detect data corruption
 - ▶ handle lost packets
- ▶ Usually implemented over IP, so TCP/IP

General outline of TCP

- ▶ TCP provides stream-like data transfer
 - ▶ By using the notion of a *connection* before transferring data
 - ▶ Each connection is identified by an address:port pair, one for both sides of a connection
 - ▶ e.g. 192.168.1.2:5810 to 8.8.8.8:80
- ▶ Delivers data in order
 - ▶ Adds sequence numbers which aid ordering
- ▶ Detect data corruption
 - ▶ Adds checksums
- ▶ Handles lost packets
 - ▶ Sequence numbers identify lost or late packets
 - ▶ Packets are retransmitted after a timeout
 - ▶ Also identifies duplicate packets
- ▶ Many other details
 - ▶ Setting up connections
 - ▶ Congestion control

Well-known ports

- ▶ TCP requires a connection to be established
 - ▶ Need to know addresses of both nodes
 - ▶ Need source port and destination port
- ▶ Many destination ports are well-known
 - ▶ 80 is for a web server using HTTP
 - ▶ 443 is for a web server using HTTPS
 - ▶ 21 is for file transfer protocol
 - ▶ 22 is for secure shell
 - ▶ 25 is for SMTP (i.e. sending email)
- ▶ Your OS usually assigns a random port to the source side of the connection

Programmer's Perspective

- ▶ Does your application need reliable, ordered communication?
 - ▶ Use TCP
- ▶ Is it transactional and can handle missing/corrupted packets?
 - ▶ Use UDP
- ▶ Note sometimes protocols mandate use of TCP or UDP
 - ▶ Domain name service (DNS) uses UDP (but also supports TCP)
 - ▶ HTTP/1 and HTTP/2 only use TCP

Outline

Administrivia

Networking

The Internet Protocol (IP)

The Sockets Client Programming Interface

The Sockets Server Programming Interface

An Echo Server and Client

Sockets

- ▶ Original terminology from ARPANET
 - ▶ Usage today very different from that definition
- ▶ A socket is one end of a connection (i.e. one address:port pair)
- ▶ Programming interface for networking applications
 - ▶ Developed and popularized by BSD Unix
- ▶ Nearly the same interface in any operating system
 - ▶ And across programming languages
 - ▶ the Plan9 OS is a notable exception

Creating sockets

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

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

- ▶ domain is AF_INET for IPv4
 - ▶ also used AF_UNIX for Inter-process Communication
- ▶ type is SOCK_STREAM or SOCK_DGRAM
 - ▶ Roughly correspond to TCP and UDP respectively
- ▶ protocol is usually 0
- ▶ socket returns a file descriptor
 - ▶ Use read and write for TCP
 - ▶ Use send and recv for UDP (can also be used for TCP)

Specifying destination

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

- ▶ For TCP, `connect` will try to open a connection to `addr`
- ▶ For UDP, `connect` will send any future packets to `addr`
 - ▶ It will also only accept packets sent by `addr`
- ▶ These must be numeric addresses, not names
 - ▶ I.e. `172.217.7.14` not `google.com`

Aside: Resolving names to addresses

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

int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- ▶ Names are “converted” into addresses using a *resolver*
 - ▶ Known as name resolution
 - ▶ Needs to look up a database of names to addresses
- ▶ Nearly always use the Domain Name Server (DNS) protocol
 - ▶ But can also use other alternatives like UPnP
- ▶ `getaddrinfo` takes a name (`node`) and a service (“http”) and returns an address
 - ▶ service can be NULL
 - ▶ Don't use `gethostbyname` which is obsolete

Sending data

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

- ▶ Sends data to a destination
 - ▶ Must have been specified using a previous connect call

Receiving data

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

- ▶ Receives data from a socket
 - ▶ This must be a connected socket
- ▶ `recv` will block until data is received

Blocking vs Nonblocking I/O

- ▶ Many functions that perform I/O block by default
 - ▶ `read`, `recv`, `scanf`
- ▶ Although this simplifies programming, programs may want to check if data is available or not
- ▶ Programs can use `fcntl` to set a socket to *non-blocking* mode
- ▶ In this case, if the operation would normally block, it will instead return `EWOULDBLOCK`
 - ▶ Or `EAGAIN`
- ▶ Allows you to overlap I/O with computation
 - ▶ Very useful prior to threading

Asynchronous I/O

- ▶ Instead of checking if an I/O operation has completed, you can use asynchronous I/O
- ▶ OS will notify your program that I/O is complete
- ▶ All operations in asynchronous I/O are non-blocking by default
- ▶ See manual page for `aio` for more information
 - ▶ Or see Overlapped I/O in Windows

Overview of Client-side Interface

- ▶ Create a socket
- ▶ connect it
- ▶ send or recv data from it
- ▶ close it

Outline

Administrivia

Networking

The Internet Protocol (IP)

The Sockets Client Programming Interface

The Sockets Server Programming Interface

An Echo Server and Client

Server-side sockets: `bind`

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

- ▶ `sockfd` is obtained from `socket`
- ▶ `addr` is *local* address to accept requests on
 - ▶ Usually set to `0.0.0.0` for IPv4 to accept requests on all of the node's IP addresses

Listen

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

- ▶ `sockfd` is a socket for which `bind` has been called
- ▶ `backlog` specifies queue depth – connections made after this has been exceeded may be refused

Accept

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

- ▶ Waits until a connection is made (unless the socket is non-blocking) to `sockfd`
 - ▶ which must be a socket on which `listen` has been called
- ▶ Returns the address of the other side of the connection in `addr`
- ▶ Returns the file descriptor for an accepted socket, i.e. the connection
 - ▶ This is used to send and receive data
 - ▶ The original socket continues to listen for more connections

Outline

Administrivia

Networking

The Internet Protocol (IP)

The Sockets Client Programming Interface

The Sockets Server Programming Interface

An Echo Server and Client

Echo Server: Creating and Binding a socket

```
l = socket(AF_INET, SOCK_STREAM, 0);
if(l == -1) {
    perror("socket");
    exit(1);
}

struct sockaddr_in addr;

memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(5000); /* convert to network byte order */
addr.sin_addr.s_addr = INADDR_ANY;

if(bind(l, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
    perror("bind");
    exit(1);
}
```

Echo Server: Listening

```
if(listen(1, 1) == -1) {  
    perror("listen");  
    exit(1);  
}
```

Echo Server: Accept Loop

```
struct sockaddr_in sender;
socklen_t addrlen = sizeof(sender);
int s, r;

char buf[256];

while((s = accept(1, (struct sockaddr *) &sender, &addrlen)) != -1) {
    r = recv(s, buf, 255, 0);
    if(r == -1) {
        perror("recv");

        close(s);
    } else {
        buf[r] = '\0';

        printf("received '%s'\n", buf);

        if(send(s, buf, r, 0) == -1) {
            perror("send");
        }

        close(s);
    }
}
```

Echo Client: Creating a socket

```
l = socket(AF_INET, SOCK_STREAM, 0);  
if(l == -1) {  
    perror("socket");  
    exit(1);  
}
```

Echo Server: Connecting

```
struct sockaddr_in addr;

memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(5000);
addr.sin_addr.s_addr = htonl(0x7f000001); // 127.0.0.1

if(connect(l, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
    perror("connect");
    exit(1);
}
printf("connected\n");
```

Echo Server: Send/Recv

```
char buf[256];
int r;

r = scanf("%255s", buf);
if(r != 1) {
    perror("scanf");
    exit(1);
}

if(send(l, buf, strlen(buf)+1, 0) == -1) {
    perror("send");
    exit(1);
}

r = recv(l, buf, 255, 0);
if(r == -1) {
    perror("recv");
    exit(1);
} else {
    buf[r] = '\0';
    printf("got '%s'\n", buf);
}
```

Problems with the Echo Server

- ▶ The echo server handles one connection at a time
- ▶ It is serial
- ▶ What if a client connects but does not send a message?
 - ▶ What happens to clients who connected after that client?

References

- ▶ Chapter 11 of the textbook
 - ▶ And Chapter 12.2 which talks about I/O multiplexing (one solution to one client at a time)
- ▶ Read the manual pages for `socket` and pages related to it
 - ▶ Or read the GNU libc manual for sockets