UMBC

Training Centers

Network Programming in C

TCPRG3004-2020-03-12

Table of Contents

Chapter 1. Introduction

The Internet is nothing but a massive network of networks. And those networks are just computers communicating.

How data can traverse such a heterogeneous network is a remarkable feat, and one worthy of study. The engineering and design that was put into the Internet is a testament to modern computer science and engineering.

Prerequisites

This material assumes an intermediate-or-better knowledge of C, and some familiarity with simple data structures. Required tools are enumerated in Chapter 2, *Tools*. Use of the Linux command-line tools for compilation will also be required.

This course defaults to using IPv6 for communication. Most systems and networking devices are now dual-stack by default, and well-made networking code will degrade gracefully from IPv6 to IPv4, but not the reverse.

The portion dealing with the Python C API also requires some background in Python.

History

All modern networking is over the Internet Protocol, or IP. Designed by Vint Cerf and Bob Kahn in 1974, it finally started taking over all networking protocols in the late 1970s.

ARPANET was an early network between universities, the military, and a few large corporations. Rather than having separate circuits for every pair of communications, ARPANET was **packet switched**, where each link between hosts could be used by arbitrary traffic. The difference is like that between a phone line where only one pair of people can talk, and a water pipe that can be used by many different appliances in a home. Each packet is labeled with its intended destination and purpose.

As more and more computers were connected to the network, the growing network became more and more useful. The Internet Protocol began taking over the earlier Host-to-Host Protocol, and the Internet was born.

The Python language was designed by Guido van Rossum in 1991, as a hobby project descended from the obscure ABC language while at the Netherlands' National Research Institute. It was designed to be readable, obvious, and extendable. This extendability today means that Python has been ported to various run-time VMs, as in Jython and IronPython, and has had much of the low-level C API exposed to curious or industrious users.

Initially gaining traction with the scientific community, Python has expanded its role to almost every possible niche in programming, from web applications to data analytics to frameworks and beyond. It is popular for use in education due to its straightforward syntax and intuitive nature.

Python engaged in a backwards-incompatible upgrade with Python 3, released in 2008. This course is concerned with Python 3, not the older Python 2 language that reaches end-of-life support in 2020.

Chapter 2. Tools

This course is designed to be run on a UNIX-like system, such as BSD or Linux. The same overall concepts also apply for Windows systems using Winsock 2, but the specific functions and types may be somewhat different.

While systems may provide different ways of interacting with higher-level protocols, this course is targeting the low-level business of network programming.

Two users on the same machine cannot bind the same port. As such, the `$UID` of each user is used to identify a port to bind to. If the user ID is too low (less that 1024), it is recommended to add 1000 to ensure that each student has a unique user ID that can be assigned as a port number.

Compiler

Any C compiler will do. Code has been tested under the `gcc` toolchain on Ubuntu Linux. Porting to other UNIX-like environments should be straightforward, but BSD-style environments might need additional changes.

For using the Python C API, CPython is assumed as a backend.

Make

This course uses GNU `make` to manage dependencies during build. Make sure that the following options are set in the Makefile:

`Makefile`

```
CFLAGS += -Wall -Wextra -Wpedantic
CFLAGS += -Wwrite-strings -Wvla -Wfloat-equal -Waggregate-return -Winline
```

These options are recommended to help catch problems in code being written.

ifconfig

`ifconfig` displays information about the network interface cards. It can also be used to see transferred/received statistics on those cards. With the right permissions, it can also be used to configure these devices, though that is outside the scope of this course.

On Linux, this tool has been deprecated in favor of the newer `ip addr` utility. To see statistics on transfers, invoke `ip -s addr`.

netstat

`netstat` prints out network statistics. It is also used to check socket and services status on the current machine.

On Linux, this tool has been deprecated in favor of newer utilities, `ip` and `ss`. `ss` will take nearly every flag that `netstat` is capable of, and runs faster.

nc

Netcat is a simple application that sends data over IP in UDP or TCP packets.

To use netcat as a client, name an endpoint (host and port) to connect to. To connect to a website, for instance, `nc example.org 80` would connect to `example.org` on port 80, the default port for HTTP traffic. By sending the right commands to the webserver, a web page can be returned.

```
$ nc example.org 80
GET / HTTP/1.1
Host: example.org

HTTP/1.1 200 OK
...
$
```

Netcat can also be used as a server via the `-l` flag, for 'listen'. By default, netcat operates on TCP traffic. To use UDP traffic, pass the `-u` flag. Finally, the `-k` flag can be used to 'keep-listening', which will keep the server alive continuously.

nmap

`nmap` is a network-mapping utility, that scans a host and determines which ports are open. While the same result can be attained using `nc -z` with a range of ports, nmap is geared towards this usage and is ultimately easier to use.

Chapter 3. Network Architecture

Any network, including the internet, is a collection of computers linked together by communication. While there are many network topologies possible, with the modern Internet, most networks involve a number of computers sharing a single communication channel, which is shared by all those computers. At least one **gateway** node on the network routes communications destined for locations outside the local network, which itself shares a communication channel with other gateways, and so on upwards.

OSI Model

The **Open Systems Interconnection** network stack is a failed competitor to the Internet. As the Internet Protocol grew more popular and widespread, major commercial companies (notably Cisco, Honeywell, and IBM) partnered with two major standards bodies to design an advanced, tiered protocol to handle networking computers.

However, OSI never came up with a deliverable implementation of the theoretical protocol suite, and Internet has been so successful that all other competing implementations (Appletalk, IPX, Netbui, DECnet, etc.) have died out.

The one useful artifact of this attempt was the abstract architecture known as the **OSI Model**.

#	Name	Example traffic
7	Application	HTTP, IRC, etc.
6	Presentation	*unused*
5	Session	*unused*
4	Transport	TCP, UDP
3	Network	IP
2	Data	Ethernet, 802.11
1	Physical	Wires, Radio Waves, Avian Carriers

As of today, layers 5 and 6 are obsolete, but the other layers are useful when broadly discussing parts of a given network packet. In reality, many of the protocols and systems use will “leak” up or down the layers.

Physical Layer

The physical layer refers to the physical media used to sustain communications. For purposes of this course, this layer is largely ignored and assumed to be free of errors.

Data Layer

The purpose of the data layer is to encode information into a form that the physical layer can move it around. This would be binary encoding onto most media, and thus is largely the Ethernet frame protocol in most networking today.

Network Layer

The goal of the network layer is to give unambiguous destinations for communication. On the Internet, this is the Internet Protocol. This is the layer that handles addressing and routing of communication.

Transport Layer

The transport layer encodes the means of disassembling and reassembling communication so that it can be successfully routed between destinations. It also is the layer that allows for reliable links and communication.

Session Layer

The idea behind the session layer was to act as verification/authentication for network traffic. So, all traffic above this layer could be encrypted, with this header providing information to the remote end on how to decrypt it.

Presentation Layer

The objective of the presentation layer is to act as a kind of translator between hosts: UTF-16 vs. UTF-8, C strings vs. Pascal strings. The upper layer specifies the data to transfer, and the presentation layer determines how to serialized that data to the network.

Application Layer

Finally, the application layer is the layer that transports the actual payload desired by the user. This is the layer that is responsible for synchronized communication between destinations.

TCP/IP Model

The **TCP/IP** network stack is shorter than the OSI model, and better reflects the current state of the Internet. It is sometimes referred to as the Department of Defense model.

Name	Example traffic
Application	HTTP, IRC, SSH, SMTP, etc.
Transport	TCP, UDP, SCTP
Internet	IP, ICMP
Network	Ethernet, ARP, OSPF

The OSI model does not reflect the certain amount of “leakage” between the layers. Rather than being fully encapsulated at each stage, the same concerns may operate at multiple levels.

For instance, the SSH protocol operates at the OSI Application layer. But, its concern is largely in the session and presentation layers, handling a connection and encryption.

The TCP/IP model more accurately reflects the Internet as it works today, especially for network programmers.

Network Layer

The network layer is responsible for encoding and decoding data for the communication media. This covers both the media itself and the encoding of data. It does not provide guarantees of in-order delivery nor delivery at all. Any such desired features must be encoded at a higher level.

Internet Layer

The internet layer handles addressing and routing of data. It can report the failure to deliver packets, diagnose some connection issues, and provides the mapping between hardware addresses and network addresses. Like the link layer, it does not provide guarantees.

Transport Layer

The transport layer provides guaranteed, in-order delivery between destinations, given a certain amount of overhead. It can also maintain the lack of guarantee for less overhead. Additionally, the transport layer is what identifies which services which to communicate at the destinations, allowing for the packet-switching structure that is the Internet.

Application Layer

The application layer provides services used by end users. It rides on top of the other layers. The majority of network programming is written at this level. Services such as the World Wide Web, email, chat, and video streaming are all provided by applications building on the lower layers for delivery.

Exercises

Exercise 1

Using Lego® bricks, build both an IPv4 and an IPv6 packet.

Chapter 4. Data Transmission

Moving Data Around

On Ethernet, to send a message, one encodes it on the wire, with the 1s and 0s of binary encoding translated to presence or absence of voltage. Any machines on the local network may listen to that message. To indicate that a message is destined for a particular host, the hardware ID, or MAC address, of the network device is set as the destination in the Ethernet frame. By convention, any frames not addressed to a host are ignored by that host. There is also a broadcast address, `FF:FF:FF:FF:FF:FF`, which all hosts listen to.

On the Internet, an IP address is designed to be a unique endpoint or node, that can be communicated with. While the IP address is just a number, it has a human-readable representation, depending on the version of the IP. For version 4, the number is written as the four bytes making up the number, each printed in decimal, separated by dots, such as `192.0.2.87`. For version 6, the number is written as pairs of bytes in hex, separated by colons, such as `2001:db8::a7:82`. Multiple IP addresses can be assigned to a given MAC.

A device can broadcast to all MAC addresses in the local subnet by having a destination of `FF:FF:FF:FF:FF:FF`. A device can broadcast to all IPv4 hosts in the local subnet by sending on the current subnet mask with all bits set. So, on the `192.0.2/24` network, the broadcast address would be `192.0.2.255`. On the `172.16.0.0/12` network, the broadcast address would be `172.31.255.255`. The subnet mask bits are fixed, and all other bits are set to one.

IPv6 does not have the concept of “broadcast”, but does have a similar concept called multicast. Any address in the subnet `ff00::/12` has a specific multicast purpose, such as to all time servers, DHCP servers, routers, etc. As an example, sending a message to the IPv6 address `ff01::2` would multicast the message to all routers local to the interface.

Given an IP address, the Internet can route a message to that host. Each network of hosts operates on a subnet mask, a fixed prefix of the IP space. If the destination has a matching prefix for the current host, then the message is sent directly on the local subnet. To determine the MAC address, the current host sends out a broadcast message, asking for the owner of that IP address to respond with their MAC address. In IPv4, this is called ARP, and in IPv6 is handled by ICMPv6 packets known as NDP.

If the destination does not have a matching subnet prefix, then the current host must use a gateway. A gateway is a device that handles traffic destined for hosts not on the local subnet. The gateway is manually configured (or semi-automatically via DHCP) in IPv4, but automatically detected via NDP broadcasts on IPv6.

Each computer and router maintains a table called the **routing table** that keeps track of which hosts are immediately addressable, and which addresses must be passed through a gateway. This information can be viewed with `ip route` (originally just `route`).

Finally, the operating system examines the rest of the headers to determine which port it is using, in the case of TCP or UDP traffic. This is then handed off as a socket to the appropriate application. The initial

port used is hard-coded by the two hosts, although they are welcome to negotiate a different set of ports to use (such as what FTP does). Ports are integers in the range 0-65,535.

Ports less than 1024 are reserved for specific protocols, such as web traffic, ssh, ftp, and the like. This does not prevent the service from operating on a different port, it is merely traditional.

DHCP

Dynamic Host Configuration Protocol allows for easy IP discovery and configuration for hosts newly connected to a network. A host with no IP address will broadcast or multicast its MAC, asking for an IP address. A DHCP server, in charge of delegating IP addresses from a pool of available ones, will assign that MAC address an IP address and respond to the requesting host. The host acknowledges receipt, and then has Internet connectivity. The DHCP lease contains information such as the IP address, subnet mask, and gateway. It can even provide DNS servers for the host to use.

Domain Names

Since IP addresses can be hard to memorize or pronounce, domain names may be used. A domain name is a period-separated series of words, ending with a top-level domain (TLD), such as `.org`, `.net`, or `.us`, for example. These are mapped to IP addresses via the Domain Name Service (DNS) protocol. Each of these TLDs maintains a server that keeps track of which domain names (such as `example`) have which DNS server addresses (such as `192.0.2.11`). A host then knows to ask this DNS server for the address of the domain name (`example.org`), receiving back its address (`192.0.2.1`).

Rather than asking a TLD for every domain name lookup, a host that is trying to resolve a domain name to an IP address asks a local DNS server, which caches requests to various TLDs. The local host in turn caches these mappings as well. The length of time to cache this information is configurable by the owner of the domain, and is transmitted along with the IP address. Even applications could hold their own DNS cache, if desired (a number of web browsers do so).

The `dig` command can be used to run these DNS queries, finding out the IP address of a given domain name.

Netcat Lab

The netcat utility can be used as a simple way to transfer arbitrary data between hosts. To see this firsthand, it will used as a web browser.

Start up netcat with `nc example.org 80`. This asks netcat to connect to the host at `example.org`, on port 80, the standard port for HTTP traffic. The connection will take place, and then netcat waits for input.

Then, type the following into netcat:

```
GET / HTTP/1.1
Host: example.org
```

Make sure that the "Host" line matches the domain being connected to. After hitting return twice, netcat will spit out a series of HTTP headers, followed by an HTML document.

Try some other hosts, and see how they respond. Some may return a redirect URL to try instead, which should replace the `/` requested after the `GET`. Some may return a redirect hostname, such as `www.example.org`.

HTTP Protocol

The HyperText Transfer Protocol (HTTP) is an Application-level protocol that enables the transmission of web resources: images, web pages, styling, and the like. Due to the ubiquity of the World Wide Web, the ease of use of this protocol, and the likelihood that ports for web traffic will be open across firewalls, many other protocols or communication schemes have been built on top of HTTP.

HTTP is built on top of TCP: This allows for in-order delivery of content, which is important when delivering text that must be parsed in a specific way, or binary data.

HTTP is designed to be simple enough for users without web browsers to use! The netcat lab is predicated on this. A TCP connection is established from client to server. The client then makes a request of the server, and the server responds with some metadata, followed by any content.

HTTP Request

The HTTP Request starts with a verb, a path, and a version. The version will always be the HTTP version (currently either 1.0 or 1.1). The path is the path (on the server) to the resource being requested. While this is often a relative path from some starting directory in which the HTTP server runs, it is not required to be. Finally, the verb in the HTTP Request is one of a subset of possible verbs or actions that can be sent to a HTTP server, such as GET, POST, PUT, DELETE, or HEAD (others exist).

Any additional lines of the request are `key: value` pairs. The only required key is a `Host` key, indicating the name of the requested server. This is used due to many web servers hosting multiple domain names. (Version 1.0 of HTTP did not have this requirement.)

Additional headers may indicate the kind of content requested, the type or capabilities of the requesting engine, timestamp information, etc. Many HTTP servers make use of these fields.

Finally, an HTTP Request is capped off with two blank lines, which tells the HTTP server that no more data will be sent.

At this point, the HTTP server processes the request. It will incorporate any or all parts of the request to produce an HTTP response.

HTTP Response

An HTTP Response is very similar to an HTTP Request, only from server to client instead of the reverse. Once the request is received and processed, the server is ready to send back the response. The HTTP Response begins with a list of `key: value` headers, just as the request did, and finally terminated with two blank lines. This is the header.

Then, any content that will be sent from the server is transmitted back to the client. The end of content is signalled either by closing the connection, or a specific `Content-Length` header field with an appropriate

value.

This is all the work that a web browser does on the network side. It requests data, then uses a display engine to render it onto a screen. But from the point of view of the network, it is straightforward.

So, too, is the web server. It has a little bit more work to do, such as potentially running scripts or resolving URLs, but it just executes the same basic tasks over and over again. This lack of complexity is common across many network servers and clients: the networking is straightforward, but what is done with the information afterwards is complex.

Exercises

Exercise 1

Using only `netcat`, issue three HTTP requests to various servers. Name 10 distinct HTTP headers that appear. Name two that exist across all servers.

Chapter 5. Sockets

Networking is communication between two endpoints. Each endpoint uniquely identifies that connection through the unique combination of source IP address, source port, destination IP address, destination port, and protocol (TCP or UDP, generally). This is also referred to as a **5-tuple** (Many descriptions refer to a 4-tuple, omitting the protocol due to it being fixed, e.g. the TCP 4-tuple). That means that to establish a connection, each of these must have some value, and that combination of values is unique to that connection.

Modern network programming involves the use of sockets. A socket is a file descriptor (`HANDLE` on Windows), and therefore just an index into a per-process table of OS resources. The **socket** one of the logical endpoints of a connection, and is therefore made up of the IP address at that end and the port that the endpoint is using. This endpoint has underlying structure that is critical to understand for networking.

This does not mean that the IP/port combo can handle only one connection at a time! The operating system maintains a table of connections, so it can correctly route communication from different hosts on the same port to the correct sockets.

A socket can be used with both `read(2)` and `write(2)`, once it is properly set up. Since there are usually a number of options involved with sockets that are not set for most file descriptors, the functions `send(2)` and `recv(2)` are usually used.

Socket Structure

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

This is a `struct sockaddr`, the interface to any socket addresses. Never create this structure directly! The `sa_family` field is like a type identifier that indicates what kind of socket (Internet, UNIX, etc.) the rest of the structure contains. This works due to C's weak typing between pointers. Based on the value of `sa_family`, the structure should be cast to the appropriate structure type. The most common values for this socket address family are `AF_INET`, `AF_INET6`, `AF_UNIX`, and `AF_PACKET` (on other systems, `AF_PACKET` may go by other names).

Storage Socket

Since a given socket address can take variable amounts of space, a special socket address known as `struct sockaddr_storage`. It has enough padding so that it can hold the largest possible `socketaddr` structure on the system. If a socket address structure must be created, and what its type will be is not known, this is the structure to use.

```
struct sockaddr_storage {
    sa_family_t ss_family;
    char pad[...];
};
```

IPv4 Socket

The standard Internet socket consists of the address and a port number. Since the socket address structure for Internet is smaller than a `struct sockaddr`, some padding is required. Both the port and the address are in network byte order.

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

The address is a `struct` due to historical reasons, allowing it to be accessed by byte, half-word, or word. These days, it is rare to interact directly with the structure, and preferred to use helper functions to fill it or

interact with it.

IPv6 Socket

Version 6 Internet sockets need to allow for the longer address field, but have the same port field as their version 4 counterpart.

```
struct sockaddr_in {
    sa_family_t sin6_family;
    uint16_t sin6_port;
    uint32_t sin6_flowinfo;
    struct in6_addr sin_addr;
    uint32_t sin6_scope_id;
};
```

The address is a `struct` that holds 16 bytes, sufficient for an IPv6 address. It is highly unusual to interact directly with these bytes; there are helper functions that convert between it and a presentation string.

UNIX Socket

UNIX domain sockets may also be used through the socket interface, at the benefit of being nearly twice as fast as Internet sockets. The important information is the path to the socket.

```
struct sockaddr_in {
    uint16_t sun_family;
    char sun_path[108];
};
```

Raw Socket

Link-layer access can be had using raw sockets. With these, the underlying MAC address of the hardware is accessible in the `sll_addr` field. Under Linux, the address family is `AF_PACKET`.

```
struct sockaddr_ll {
    unsigned short sll_family;
    unsigned short sll_protocol;
    int            sll_ifindex;
    unsigned short sll_hatype;
    unsigned char  sll_pkttype;
    unsigned char  sll_halen;
    unsigned char  sll_addr[8];
};
```

Root-level permissions are needed to create raw sockets. But, they can be used to write out network data that does not get packed into TCP, UDP, or even IP headers.

Interface List

With these socket address types, it is possible to make a program that displays a listing of every interface on the system, using the function `getifaddrs(3)`.

if_list.c

```
#include <ifaddrs.h>
#include <stdio.h>
#include <sysexits.h>

#include <arpa/inet.h>

#include <linux/if_packet.h>

#include <sys/types.h>
#include <sys/un.h>

int main(void)
{
    struct ifaddrs *results;
    int err = getifaddrs(&results);
    if(err != 0) {
        perror("Could not retrieve interfaces");
        return EX_UNAVAILABLE;
    }

    for(struct ifaddrs *p = results; p; p = p->ifa_next) {
        printf("%s\t", p->ifa_name);

        char addr[INET6_ADDRSTRLEN];
        struct sockaddr *sa = p->ifa_addr;
        if(sa->sa_family == AF_INET6) {
            inet_ntop(sa->sa_family, &((struct sockaddr_in6 *)sa)->sin6_addr, addr,
sizeof(addr));
            printf("IPv6 %s ", addr);
        } else if(sa->sa_family == AF_INET) {
            inet_ntop(sa->sa_family, &((struct sockaddr_in *)sa)->sin_addr, addr, sizeof
(addr));
            printf("IPv4 %s ", addr);
        } else if(sa->sa_family == AF_UNIX) {
            printf("UNIX %s", ((struct sockaddr_un *)sa)->sun_path);
        } else if(sa->sa_family == AF_PACKET) {
            uint8_t *ptr = (uint8_t *)((struct sockaddr_ll *)sa)->sll_addr;
            printf("LINK %02x:%02x:%02x:%02x:%02x:%02x ", ptr[0], ptr[1], ptr[2], ptr[3],
ptr[4], ptr[5]);
        }

        puts("");
    }

    freeifaddrs(results);
}
```

A number of new functions and constants are introduced here.

`inet_ntop` takes a structure in network order and converts it to presentation, i.e., a string. Note that it takes the address family type, and writes the string to the passed-in buffer. `INET6_ADDRSTRLEN` is a constant that is the maximum length of an IPv6 string, plus its NUL byte.

Since `getifaddrs(3)` allocates memory, that memory must be freed via `freeifaddrs(3)`. The structure returned is a linked list that contains `struct sockaddr` members. The amount of pointer casting is extremely common when writing networking code.

The output of this program shows which network interface devices exist on the current machine, and what addresses are assigned to them.

Functions to Avoid

Do **not** use `inet_aton`, `inet_ntoa`, `gethostby*`, `getservby*`. These are part of the deprecated BSD Socket API. Use `inet_pton`, `inet_ntop`, `getaddrinfo`, and `getnameinfo`. Instead of "ASCII to Network", the mnemonic is "Presentation to Network" and vice versa.

Reverse DNS Lookup

Given an IP address, it is often possible to ask the DNS service what domain name is associated with it.

reverse_dns.c

```
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <sysexits.h>

#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    if(argc != 2) {
        fprintf(stderr, "Usage: %s <IP>\n", argv[0]);
        return EX_USAGE;
    }

    struct sockaddr_storage address = {0};
    if(strchr(argv[1], ':')) {
        address.ss_family = AF_INET6;
        int err = inet_pton(address.ss_family, argv[1], &((struct sockaddr_in6 *)&
address)->sin6_addr);
        if(err < 0) {
            perror("Could not parse address");
            return EX_USAGE;
        } else if(err < 1) {
            fprintf(stderr, "Could not parse IPv6 address\n");
            return EX_USAGE;
        }
    } else {
        address.ss_family = AF_INET;
        int err = inet_pton(address.ss_family, argv[1], &((struct sockaddr_in *)&address)
->sin_addr);
        if(err < 0) {
            perror("Could not parse address");
            return EX_USAGE;
        } else if(err < 1) {
            fprintf(stderr, "Could not parse IPv4 address\n");
            return EX_USAGE;
        }
    }

    // static because this is otherwise a huge buffer to put on the stack
    static char host[NI_MAXHOST];
```

```
    int err = getnameinfo((struct sockaddr *)&address, sizeof(address), host, sizeof(
host), NULL, 0, 0);
    if(err != 0) {
        fprintf(stderr, "Cannot get DNS: %s\n", gai_strerror(err));
        return EX_UNAVAILABLE;
    }

    printf("%s\n", host);
}
```

The program packs a `struct sockaddr_storage` with the given IP address (in network byte order), via the `inet_pton(3)` function. The function `getnameinfo(3)` does the actual work of looking up the domain name information for the network endpoint. It is also possible to look up service names (FTP, etc.) using this function, if desired.

IP List

ip_lookup.c

```c
#include <netdb.h>
#include <stdio.h>
#include <sysexits.h>

#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    if(argc != 2) {
        fprintf(stderr, "Usage: %s <host>\n", argv[0]);
        return EX_USAGE;
    }

    struct addrinfo hints = {0};
    hints.ai_family = PF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    struct addrinfo *results;
    int err = getaddrinfo(argv[1], NULL, &hints, &results);
    if(err != 0) {
        fprintf(stderr, "Cannot get address: %s\n", gai_strerror(err));
        return EX_USAGE;
    }

    printf("IP Addresses for %s\n", argv[1]);
    for(struct addrinfo *p=results; p; p = p->ai_next) {
        void *addr;

        if(p->ai_family == PF_INET6) {
            struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
            addr = &(ipv6->sin6_addr);
        } else {
            struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
            addr = &(ipv4->sin_addr);
        }

        char ipstr[INET6_ADDRSTRLEN];
        inet_ntop(p->ai_family, addr, ipstr, sizeof(ipstr));
        printf("  %s\n", ipstr);
    }

    freeaddrinfo(results);
}
```

This program provides a list of all IP addresses associated with a given host. It introduces one of the most

important helper functions in modern network programming, `getaddrinfo(3)`.

Given a host and port, it builds the appropriate endpoint data and allocates it into `results`. This function may optionally take a similar structure hinting as to the desired results.

```
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    socklen_t ai_addrlen;
    struct sockaddr ai_addr;
    char *ai_canonname;
    struct addrinfo *ai_next;
};
```

The results returned are a linked list of every endpoint that could be discovered for the requested host and port. Since the function allocates memory, it must later be freed with `freeaddrinfo(3)`. Any errors that occur when calling `getaddrinfo(3)` may be reported via the `gai_strerror(3)` function.

This function is more convenient and less error-prone than manually packing a `struct sockaddr` to be used in a call to `sendto(2)`.

The `getaddrinfo(3)` function may perform a DNS lookup, mapping a domain name to an IP address. It does this by asking the operating system to do the DNS query, and using the result. This may result in its own set of TCP `connect(2)-send(2)-recv(2)` call chain. This can be prevented with the `AI_NUMERICHOST` flag being set in the `hints` structure.

In this program, the `hints` are set to ask for either version of IP address via `PF_UNSPEC`, and to only retrieve streaming connections with `SOCK_STREAM`. If this was not specified, each IP address would be duplicated, for both streaming and datagram connections.

TCP Connections

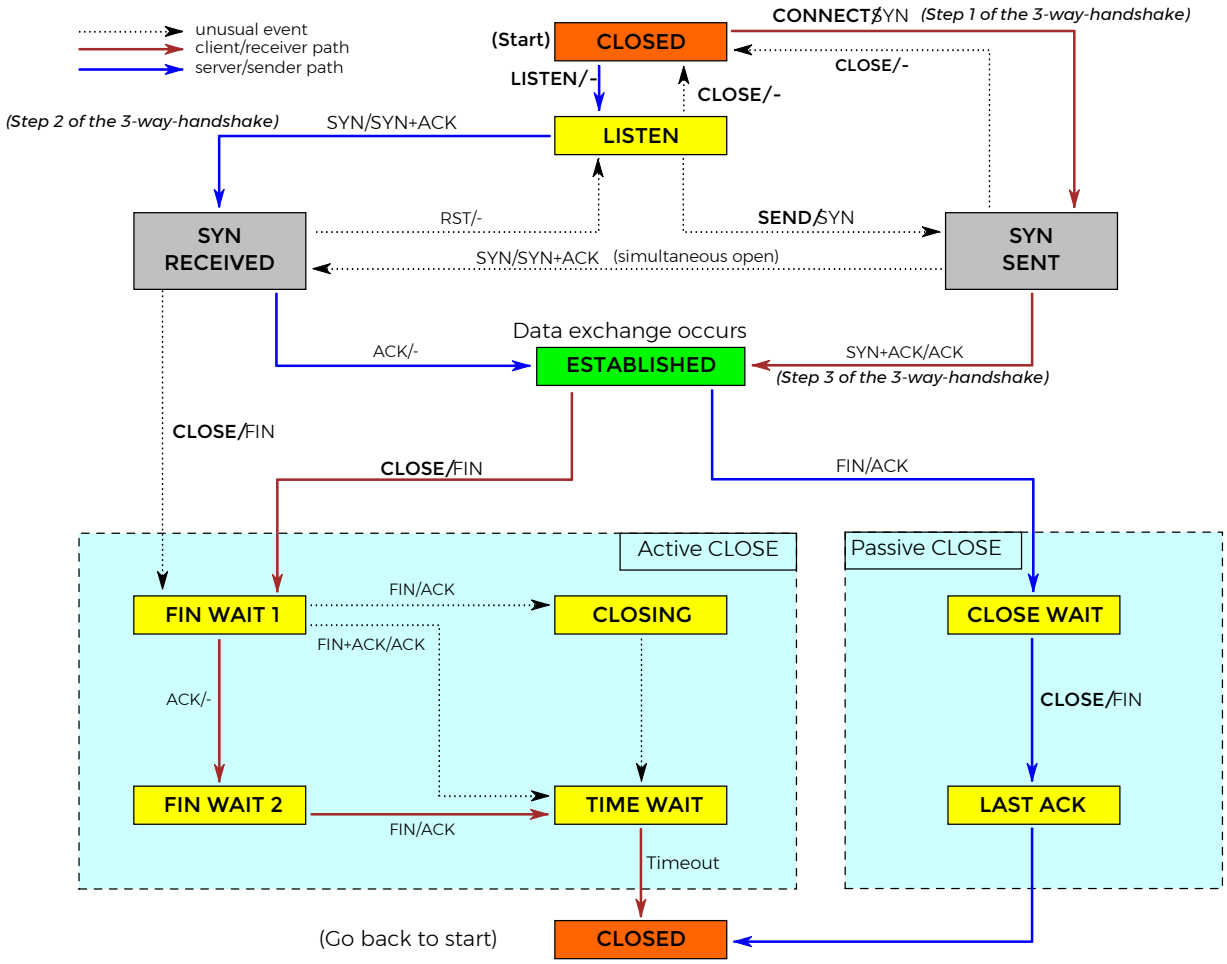

Figure 1. TCP State Diagram, Marty Pauley, Creative Commons 3.0 License

Three-Way Connecting Handshake

TCP starts its connection with a listening endpoint and a client endpoint. The client, also known as the active opener, starts out with sending a SYN packet (a packet with the SYN flag set). The listener (passive opener) responds with a SYN-ACK packet, synchronizing its end of the connection and acknowledging the client's connection attempt. Finally, when the client responds with a final ACK packet, the connection is established and normal communications can begin. This process is known as the **three-way handshake**, or SYN-SYNACK-ACK.

Four-Way Closing Handshake

Either end of a TCP connection can initiate the process of shutting down the connection. This active closer has to take five steps:

- Send a FIN to the other end of the socket (FIN_WAIT_1)

- Receive the ACK from the other end (FIN_WAIT_2)
- Receive the FIN from the other end
- Send an ACK to the other end (TIME_WAIT)
- Wait for double the Maximum Segment Lifetime (CLOSED)

As the last step is just waiting, this is called the **four-way handshake**, as four packets are sent to close the connection.

Note that it is possible for step 3 to occur at the same time as step 1! If both ends of the connection decide to terminate it, both ends wait for the confirmation ACK before moving to the TIME_WAIT state.

More likely, however, is that just one endpoint initiates the shutdown.

TIME_WAIT

One of the more confusing parts of closing a TCP connection is the TIME_WAIT state. This is a special state where the initiator of the closing waits to make sure of two things:

- That its final ACK was received.
- That any misrouted packets have expired.

What this means is that for up to four minutes, the OS may keep this endpoint open, and thus **prevent** new sockets from being created with the same IP and port. For ephemeral ports, this is not an issue. But a server may be unable to restart on a given port, failing with the message `Address already in use`. This can be disabled via the `SO_REUSEADDR` socket option, explained later.

Signal Handling

Since so much networking code is at the level of system calls, it is vital to understand how signals interrupt and restart those calls. Especially so since servers tend to be multiplex, so skill with signals are required to prevent wedges, leaks, or corrupted data.

When calling `connect(2)` in a blocking manner, other signals will interrupt the connection without restarting it (`SA_RESTART`), which means the SYN may have already been sent, but the connection is not established properly. `poll(2)` can be used to test the socket correctly.

Thus, using `signal(3)` will automatically set the `SA_RESTART` flag on that signal's action. So, the system call will restart rather than be cancelled. The flag must be removed by using `sigaction(2)` instead of `signal(3)`.

Weirdly, `accept(2)` **will** restart after receiving most signals!

Socket Options

A socket has a number of options that can be set for that specific socket. These control how the connection behaves in certain situations. This information can be retrieved using the `getsockopt(2)` call. These settings can be set with the `setsockopt(2)`. Each option operates at a specific "level" of the socket, from the socket itself, to the internet layer, to the transport layer. Socket-level options are configured with the `SOL_SOCKET` level. Internet-level options are configured with `IPPROTO_IP`. The transport-level options are configured with either `IPPROTO_TCP` or `IPPROTO_UDP`.

TCP_NODELAY

TCP has a built-in algorithm known as **Nagle's Algorithm**. The goal of this algorithm is to ensure that only one packet will be in transit between two endpoints. Since each TCP packet sent requires an ACK to be sent back, this can create a lot of congestion with small TCP packets. Nagle's Algorithm will prevent a packet from being sent until either all ACKs are received, or the amount of data has been buffered up to be the maximum segment size.

This waiting can be disabled by setting the `TCP_NODELAY` socket option on the socket. Note that it should be set on the sending side of a connection. Disabling this waiting can result in an decrease in latency, at the cost of overall throughput.

TCP_QUICKACK

A local 40ms delay should arouse high suspicion. This is the Linux TCP ACK delay! The goal of delayed ACKs is designed to reduce network congestion for telnet sessions, by combining multiple received packets' ACKs into a single response. Unfortunately, that means that a given receiver might wait up to this delay (which is configurable up to 500ms) before responding with the ACK. When combined with Nagle's Algorthim of delaying a send until all ACKs have been received, this can lead to large timeouts between two TCP endpoints. Instead, it is generally beneficial to send an ACK for a packet as soon as possible.

The combination of Nagle's Algorthim and Delayed ACK interact together very poorly from a performance perspective. It is like having a water pipe that has large pockets of air as well as water. Because of the air, the water pipe has lost its effective pressure and cannot move the water along as efficiently.

Disabling delayed ACKs can be achieved by setting the `TCP_QUICKACK` socket option on the socket. This is not as portable as `TCP_NODELAY`, but is gaining traction on other operating systems. Note that it should be set on the receiving side of a connection.

SO_REUSEADDR

At TCP termination, the initiator of the close will enter the TIME_WAIT state, waiting for any lost duplicate packets to expire. The operating system sees these endpoints as "occupied", and will not allow that

endpoint to be reused until this timeout has expired, which is on the order of minutes. To allow the operating system to reuse this endpoint, the `SO_REUSEADDR` socket option should be set.

Every server should set this socket option.

SO_LINGER

TCP goes through a termination handshake (FIN-ACK-FIN-ACK), just as it goes through a intialization handshake (SYN-SYNACK-ACK). This termination allows for confirmation that all sent data has been received by the remote endpoint. This can be disabled via unsetting the `SO_LINGER` socket option on the socket. It is possible that disabling this clean shutdown of the socket can result in fewer TCP connections in the FIN_WAIT2 state, increasing performance.

SO_KEEPALIVE

In some situations, various routers or switches may drop connections that are inactive for a certain period of time. To prevent this, the socket option `SO_KEEPALIVE` may be set on the socket. This will send a PSH packet after a configurable amount of inactive time on the socket. This will result in either an ACK (the connection still exists on the remote), or a RST (the connection has gone down on the remote, and the local end of the socket may be closed).

The amount of time to wait is configurable with `TCP_KEEPIDLE` (`TCP_KEEPINTVL` is how long to wait to resend the PSH after the first one, and `TCP_KEEPCNT` is how many retries to attempt).

Dual Stack

IPv4 and IPv6 have different protocols backed by different underlying data structures (`struct`s `sockaddr_in` and `sockaddr_in6`). These two different protocols operate at a low level; this means that while higher-level communications may be indifferent to the internet layer used, the code written needs to know which type of structure to use.

Since IPv6 is designed to be the successor to IPv4, a number of translations to allow backwards compatibility have been made, that allow IPv4 traffic over IPv6. This means that a server that is offering IPv6 service can, when set up correctly, receive connections from both IPv4 and IPv6 clients.

Exercises

Exercise 1

Name three other socket options not mentioned in this chapter.

Exercise 2

What error or warning is generated if the `struct sockaddr *` casts are removed from the sample code?

Exercise 3

Write a program that, when passed a port number, prints the typical service name associated with that number (e.g., "80" should yield "http"). Use the `getnameinfo` function to extract this information.

Exercise 4

Write a program that prints out the size of each socket structure.

Chapter 6. Uniplex Clients and Servers

UDP Client

The simplest form of network communication is a client that sends data over UDP to a destination. It follows these steps:

- Pack Destination Object
- Create Socket
- Send Data to Destination
- Close Socket

send_gai_udp.c

```c
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <sysexits.h>
#include <unistd.h>

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

int main(int argc, char *argv[])
{
    if(argc != 4) {
        fprintf(stderr, "Usage: %s <dest-ip> <port> <message>\n", argv[0]);
        return EX_USAGE;
    }

    struct addrinfo *results;
    int err = getaddrinfo(argv[1], argv[2], NULL, &results);
    if(err != 0) {
        fprintf(stderr, "Cannot get address: %s\n", gai_strerror(err));
        return EX_NOHOST;
    }

    int sd = socket(results->ai_family, SOCK_DGRAM, 0);
    if(sd < 0) {
        perror("Could not create socket");
        freeaddrinfo(results);
        return EX_OSERR;
    }

    ssize_t sent = sendto(sd, argv[3], strlen(argv[3]), 0,
            results->ai_addr, results->ai_addrlen);
    if(sent < 0) {
        perror("Unable to send");
        close(sd);
        freeaddrinfo(results);
        return EX_UNAVAILABLE;
    }

    close(sd);
    freeaddrinfo(results);
}
```

On the same machine, `nc -u -l localhost $UID` will start netcat. Then start the program with the arguments `localhost $UID Hello`. The netcat session will show the message as received.

This program introduces quite a few new functions. The most important is the `socket(2)` call, which creates the socket. This needs a protocol family (`PF_INET`, `PF_INET6`, `PF_UNIX`, or `PF_PACKET`), a socket type (`SOCK_DGRAM`, `SOCK_STREAM`, or `SOCK_RAW`), and optionally, a protocol. It returns a socket, which is a file descriptor. Therefore, the socket must be released via `close(2)`, just like a file descriptor. Just like other file descriptors, the OS will close them automatically and the end of the program.

Note that the first argument is a protocol family, not an address family. While this may sound ominous, the constants for protocol and address families have the same values.

The `sendto(2)` function sends a datagram payload through a socket to a remote endpoint. Like `write(2)`, it needs both the payload and its length. It also takes an flags argument, as well as both the `struct sockaddr` for the remote endpoint, and the size of the endpoint's structure.

TCP Client

Streaming communication is done over TCP. It is largely the same as UDP in terms of code, with one additional function call on the client side.

On the same machine, start netcat with `nc -l localhost $UID`. Then start the client program with the arguments `localhost $UID Hello`. The netcat session will show the message as received, and then exit.

The additional function call is a `connect(2)` call, which connects to the remote endpoint specified by its arguments. Note carefully that those `connect(2)` arguments are identical to the latter arguments of `sendto(2)`. Now that this connection is established, the `send(2)` call may be used to communicate with that endpoint.

A TCP connection is exactly that: a connection. Thus, it requires a call to `connect(2)` before transferring data, unlike the connectionless datagrams of UDP.

`send_gai_tcp.c`

```
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <sysexits.h>
#include <unistd.h>

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

int main(int argc, char *argv[])
{
```

```
    if(argc != 4) {
        fprintf(stderr, "Usage: %s <dest-ip> <port> <message>\n", argv[0]);
        return EX_USAGE;
    }

    struct addrinfo *results;
    int err = getaddrinfo(argv[1], argv[2], NULL, &results);
    if(err != 0) {
        fprintf(stderr, "Cannot get address: %s\n", gai_strerror(err));
        return EX_NOHOST;
    }

    int sd = socket(results->ai_family, SOCK_STREAM, 0);
    if(sd < 0) {
        perror("Could not create socket");
        freeaddrinfo(results);
        return EX_OSERR;
    }

    err = connect(sd, results->ai_addr, results->ai_addrlen);
    if(err < 0) {
        perror("Could not connect to remote");
        close(sd);
        freeaddrinfo(results);
        return EX_UNAVAILABLE;
    }

    ssize_t sent = send(sd, argv[3], strlen(argv[3]), 0);
    if(sent < 0) {
        perror("Unable to send");
        close(sd);
        freeaddrinfo(results);
        return EX_UNAVAILABLE;
    }

    close(sd);
    freeaddrinfo(results);
}
```

UDP Server

Operating a server, rather than a client, requires a different architecture. The server must be ready to receive datagrams continuously until the server is terminated. It introduces two functions that are used for a network server using connectionless datagrams.

recv_gai_udp.c

```c
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <sysexits.h>
#include <unistd.h>

#include <arpa/inet.h>

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

int main(int argc, char *argv[])
{
    if(argc != 2) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        return EX_USAGE;
    }

    struct addrinfo hints = {0};
    hints.ai_family = PF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE;
    struct addrinfo *results;
    int err = getaddrinfo(NULL, argv[1], &hints, &results);
    if(err != 0) {
        fprintf(stderr, "Cannot get address: %s\n", gai_strerror(err));
        return EX_NOHOST;
    }

    int sd = socket(results->ai_family, results->ai_socktype, results->ai_protocol);
    if(sd < 0) {
        perror("Could not create socket");
        freeaddrinfo(results);
        return EX_OSERR;
    }

    err = bind(sd, results->ai_addr, results->ai_addrlen);
    if(err < 0) {
        perror("Could not bind socket");
        close(sd);
        freeaddrinfo(results);
        return EX_OSERR;
    }
    freeaddrinfo(results);
```

- Pack Server Object
- Create Socket
- Bind Socket to Endpoint
- Endlessly Receive Data from Destination

recv_gai_udp.c

```
    for(;;) {
        struct sockaddr_storage client;
        socklen_t client_sz = sizeof(client);
        char addr[INET6_ADDRSTRLEN];

        // Internet minimum dgram size is 576, so round down to nearest power of 2
        char buffer[512];
        ssize_t received = recvfrom(sd, buffer, sizeof(buffer)-1, 0,
                (struct sockaddr *)&client, &client_sz);
        if(received < 0) {
            perror("Unable to receive");
            close(sd);
            return EX_UNAVAILABLE;
        }
        buffer[received] = '\0';

        unsigned short port = 0;
        if(client.ss_family == AF_INET6) {
            inet_ntop(client.ss_family,
                    &((struct sockaddr_in6 *)&client)->sin6_addr,
                    addr, sizeof(addr));
            port = ntohs(((struct sockaddr_in6 *)&client)->sin6_port);
        } else {
            inet_ntop(client.ss_family,
                    &((struct sockaddr_in *)&client)->sin_addr,
                    addr, sizeof(addr));
            port = ntohs(((struct sockaddr_in *)&client)->sin_port);
        }
        printf("Received from %s:%hu\n%s\n\n", addr, port, buffer);
    }

    close(sd);
}
```

The first new function introduced here is the `bind(2)` call. This requests the operating system to allow the server program to bind itself to a endpoint (in this case, the port set on the command line). Without this function, the endpoint is assigned a random port in the "emphemeral port" range. So, even a client program is allowed to call `bind(2)`, but this is very atypical.

The second new function is `recvfrom(2)`. Like `sendto(2)`, it uses a buffer, flags, and a socket address structure. It differs in that the `struct sockaddr` and its length are written to, rather than read. They are filled in the values of the remote endpoint, the client.

The rest of the function calls have been introduced before, but notice the size of the receive buffer. Datagrams sent through the Internet can technically be of any size from 68 bytes to 4GiB, but each endpoint must be able to receive a datagram of at least 576 bytes (RFC 791, p. 24). In reality, any UDP service will have a defined format that it uses and can set its buffer size to that appropriate format's size.

Since this program is designed to demonstrate a server accepting messages from remote clients, an arbitrary buffer size was chosen. As this is a datagram service, any additional data in the payload will be dropped by the operating system when the `recvfrom(2)` call completes.

Uniplex TCP Server

Next is a TCP server that services one request at a time. This is highly unusual, done to introduce concepts at a reasonable pace.

`recv_1_tcp.c`

```
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <sysexits.h>
#include <unistd.h>

#include <arpa/inet.h>

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

int main(int argc, char *argv[])
{
    if(argc != 2) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        return EX_USAGE;
    }

    struct addrinfo hints = {0};
    hints.ai_family = PF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    struct addrinfo *results;
    int err = getaddrinfo(NULL, argv[1], &hints, &results);
    if(err != 0) {
        fprintf(stderr, "Cannot get address: %s\n", gai_strerror(err));
```

```
        return EX_NOHOST;
    }

    int sd = socket(results->ai_family, results->ai_socktype, results->ai_protocol);
    if(sd < 0) {
        perror("Could not create socket");
        freeaddrinfo(results);
        return EX_OSERR;
    }

    err = bind(sd, results->ai_addr, results->ai_addrlen);
    if(err < 0) {
        perror("Could not bind socket");
        close(sd);
        freeaddrinfo(results);
        return EX_OSERR;
    }
    freeaddrinfo(results);

    // Backlog of 5 is typical
    err = listen(sd, 5);
    if(err < 0) {
        perror("Could not listen on socket");
        close(sd);
        return EX_OSERR;
```

When it comes to setting up the server socket, the major difference for TCP is that after calling `bind(2)`, the new function `listen(2)` is called on that socket, with a backlog parameter. This informs the operating system that the program wishes to establish streaming connections with any clients that connect to the socket, and that the operating system will queue up to `backlog` amount. In reality, most operating systems maintain a backlog of up to double the specified amount.

recv_1_tcp.c

```
    for(;;) {
        struct sockaddr_storage client;
        socklen_t client_sz = sizeof(client);
        char addr[INET6_ADDRSTRLEN];

        int remote = accept(sd, (struct sockaddr *)&client, &client_sz);

        unsigned short port = 0;
        if(client.ss_family == AF_INET6) {
            inet_ntop(client.ss_family,
                    &((struct sockaddr_in6 *)&client)->sin6_addr, addr, sizeof(addr));
            port = ntohs(((struct sockaddr_in6 *)&client)->sin6_port);
        } else {
            inet_ntop(client.ss_family, &((struct sockaddr_in *)&client)->sin_addr,
                    addr, sizeof(addr));
            port = ntohs(((struct sockaddr_in *)&client)->sin_port);
        }
        printf("Received from %s:%hu\n", addr, port);

        // Use a small buffer for such a demo program
        char buffer[128];
        ssize_t received = recv(remote, buffer, sizeof(buffer)-1, 0);
        while(received > 0) {
            buffer[received] = '\0';
            printf("%s", buffer);
            received = recv(remote, buffer, sizeof(buffer)-1, 0);
        }
        if(received < 0) {
            perror("Unable to receive");
        }

        close(remote);
        puts("");
    }

    close(sd);
}
```

During the infinite loop, a greater change between TCP and UDP is visible. Each remote connection is first accepted from the operating system via the `accept(2)` call. This call fills in the `struct sockaddr` structure and size of the remote endpoint, which was fully handled by `recvfrom(2)` in the UDP example.

This `accept(2)` call returns a new socket descriptor! This is what will allow for the multiplexed TCP server in the future: Every connection will be assigned its own socket descriptor by the operating system.

The `recv(2)` call receives data from the listening socket, returning the amount of data received. When the socket is closed at the other end, the call returns 0. Hence the loop exiting when `recv(2)` finally returns 0.

Exercises

Exercise 1

Implement a `datetime` server in UDP. You may need information from RFC 867 to complete this work.

Exercise 2

Implement a `discard` server in both TCP and UDP (as separate programs). You may need information from RFC 863 to complete this work.

Chapter 7. Multiplexing

Multiplex TCP Server

When it comes to multiplexing network connections, the server must use a parallel approach.

recv_n_tcp.c

```c
#include <netdb.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sysexits.h>
#include <unistd.h>

#include <arpa/inet.h>

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

int main(int argc, char *argv[])
{
    if(argc != 2) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        return EX_USAGE;
    }

    struct addrinfo hints = {0};
    hints.ai_family = PF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    struct addrinfo *results;
    int err = getaddrinfo(NULL, argv[1], &hints, &results);
    if(err != 0) {
        fprintf(stderr, "Cannot get address: %s\n", gai_strerror(err));
        return EX_NOHOST;
    }

    int sd = socket(results->ai_family, results->ai_socktype, results->ai_protocol);
    if(sd < 0) {
        perror("Could not create socket");
        freeaddrinfo(results);
        return EX_OSERR;
    }
```

recv_n_tcp.c

```
    err = bind(sd, results->ai_addr, results->ai_addrlen);
    if(err < 0) {
        perror("Could not bind socket");
        close(sd);
        freeaddrinfo(results);
        return EX_OSERR;
    }
    freeaddrinfo(results);

    // Backlog of 5 is typical
    err = listen(sd, 5);
    if(err < 0) {
        perror("Could not listen on socket");
        close(sd);
        return EX_OSERR;
    }

    struct sigaction ignorer = {0};
    ignorer.sa_handler = SIG_IGN;
    err = sigaction(SIGCHLD, &ignorer, NULL);
    if(err < 0) {
        perror("Could not ignore children completion");
        close(sd);
        return EX_OSERR;
    }
```

recv_n_tcp.c

```
    for(;;) {
        struct sockaddr_storage client;
        socklen_t client_sz = sizeof(client);

        int remote = accept(sd, (struct sockaddr *)&client, &client_sz);
        if(remote < 0) {
            perror("Could not accept remote");
            continue;
        }

        pid_t child = fork();

        if(child == 0) {
            close(sd);

            char addr[INET6_ADDRSTRLEN];
            unsigned short port = 0;
            if(client.ss_family == AF_INET6) {
                inet_ntop(client.ss_family,
                        &((struct sockaddr_in6 *)&client)->sin6_addr,
                        addr, sizeof(addr));
                port = ntohs(((struct sockaddr_in6 *)&client)->sin6_port);
            } else {
                inet_ntop(client.ss_family,
                        &((struct sockaddr_in *)&client)->sin_addr,
                        addr, sizeof(addr));
                port = ntohs(((struct sockaddr_in *)&client)->sin_port);
            }
            printf("Received from %s:%hu\n", addr, port);

            // Use a small buffer for such a demo program
            char buffer[128];
            ssize_t received = recv(remote, buffer, sizeof(buffer)-1, 0);
            while(received > 0) {
                buffer[received] = '\0';
                printf("%s", buffer);
                received = recv(remote, buffer, sizeof(buffer)-1, 0);
            }
            if(received < 0) {
                perror("Unable to receive");
            }

            close(remote);
            puts("");

            return 0;
```

```
        } else if(child < 0) {
            perror("Could not fork");
        }

        close(remote);
    }
```

The approach shown is uses forked processes to handle each connection. Since the exit from each child is discarded, the `SIGCHLD` action is to ignore any child signals (if left in the default handler, the children would become zombies, consuming resources without dying).

Inside the child process, the first step is to close the listening socket. This is not required, but a good idea. It prevents accidental reads or accepts from the listening socket on the part of the child.

It is vital that the child exit when it is complete. It would be very easy to run out of system resources if children are allowed build up without exiting.

On the server side, the remote connection is closed. Note that this will happen even if there is a error in forking the subprocess. Like any file descriptor, the `close(2)` call reduces the global refcount of the descriptor in question. This is why the connection (the same descriptor) can be closed in the parent but still be active in the child process. It is not until the refcount drops to 0 (via both processes calling `close(2)`) that the socket is closed. In the case of a TCP socket, however, there is one additional complication.

`select` and `poll`

When working with multiple streams of incoming data, it is important to service them as efficiently as possible. The `poll(2)` call accepts a collections of file descriptors, and events that the system should listen for on those file descriptors.

A similar call, `select(2)`, is the older version of finding a waiting file descriptor. However, it is a bit harder to set up an appropriate call. Rather than accepting an argument that indicates the number of file descriptors to monitor, the **highest-numbered** file descriptor is passed instead. Given a compact list of descriptors, it is theoretically possible for this to be faster than a similar call to `poll(2)`. But an additional drawback of `select(2)` is that the arguments may be overwritten during the call, and thus need to be rebuilt prior to every call to `select(2)`. These two drawbacks generally make `select(2)` a poor choice when `poll(2)` is available (some very arcane systems may only have `select(2)` available, but not `poll(2)`).

Naturally, working with multiple file descriptors requires a concurrent design to the program. The program should be designed in such a way that regardless of which descriptor is available, in whatever order, the program should continue to function normally.

Exercises

Exercise 1

Write the `chargen` service in TCP. You may need information from RFC 864 to complete this task. Why is this service generally being phased out by sysadmins?

Exercise 2

Write the `finder` service in TCP. You may need information from RFC 1288 to complete this task. Why is this service generally being phased out by sysadmins?

Exercise 3

Write a program `recurse-search` which, given a directory and a text string, recursively searches that directory tree for the first file containing that string, then exiting. Use parallellized architecture.

Chapter 8. Client/Server Architecture

Thus far, all discussion of interaction between hosts or machines has been that of a single server servicing some number of clients. Generally, the server is started, then clients connect to the server and either start receiving data immediately, or start sending data immediately.

In a simplex or uniplex architecture, only one client is being serviced at a time. The server devotes all of its resources to that one client. For very simple UDP services, this may be perfectly adequate. Similarly, for large two-node communications, this may be acceptable.

That assumption breaks down when long-lived connections are possible (TCP), or when massive numbers of clients need to be served simultaneously.

Synchronous Communications

For particularly long-lived connections, a pattern will likely be established between the two endpoints. Such a connection is likely referred to as **synchronous**. In such a setup, by convention, one endpoint talks while the other listens. For instance, with a web server, the client makes a full request, then the server responds to the request. By convention, each endpoint takes a turn in the process, with neither end "talking over" the other.

But in a peer-to-peer example, there may be two endpoints connected that could produce the same kind of traffic at the same time, communicating simultaneously. As an example, a sharded database could have two shards accepting updates, then passing those updates along to the other. Both have valid pieces of traffic that need to get through, but it is not immediately clear which has priority without additional infrastructure (since there is no explicit convention as in a web server).

As expected, synchronous channels of communication are generally over TCP, where the reliability and in-order guarantee provide a solid underpinning to any network tasks.

Asynchronous Communications

During asynchronous communications, messages may appear at any time in any order. While it seems like UDP is the natural fit for this kind of approach, any established TCP connection may also work in this manner, such as the earlier example of a sharded database. The main design consideration is when the communications may start and stop unexpectedly between the two endpoints.

TCP might be used asynchronously, for example, if in-order delivery was required, or “guaranteed” delivery.

In such a case, it is important to multiplex the incoming and outgoing data streams. Spin-waiting for input is a surefire way to deadlock the system.

Exercises

Exercise 1

Write the `echo` service in TCP. You may need information from RFC 862 to complete this task. Is this service synchronous or asynchronous? Why?

Chapter 9. Worker Pools

When it comes to dealing with large numbers of clients or requests, it can be useful for a server to have an initialized, ready pool of workers to accomodate requests, rather than forking a new process or spawning a new thread every time.

With this form of architecture, there is usually some communal pool of tasks that a set number of workers pull jobs from, even though the term is "worker" pool, not "task" pool.

While this discussion is possible in terms of multiplexing communications and parallelism/concurrency, it makes sense to discuss it in context of client/server communications.

To start with, it is assumed that these worker pools are in some form of Producer/Consumer relationship. Either multiple workers are producing, or multiple workers are consuming, or both.

The main reason for creating a worker pool is to avoid the overhead involved in process/thread creation for multiple (usually small) tasks. As long as all workers are kept relatively active (and balanced), worker pools can yield significant benefits, at the cost of increased development complexity.

Scheduling

Since each member of a worker pool tend to be independent and identical, inter-thread communication is usually not an issue. However, scheduling and managing the workers themselves requires a certain amount of overhead and scheduling.

This is generally going to involve some form of semaphore, which dictates how many active workers are running in the worker pool. Each of them will pull from some producer (usually a queue or heap), and start their work. Once the work is complete, they hand the completed task over to another queue or heap for any additional processing needed.

Many higher-level languages (such as Python or Go) expose Worker Pools as high-level classes or constructs. In C, a Worker Pool would need to be built manually, using semaphores and mutexes. Finally, some sort of signal (either a set of "finish" jobs, or some semaphore tracking which workers are running) would need to be sent to the workers for them to exit gracefully.

```
    thread_safe_queue q = tsq_init();
    tsq_add_job(q, puts, "1");
    tsq_add_job(q, puts, "2");
    tsq_add_job(q, puts, "3");
    tsq_add_job(q, puts, "4");
    tsq_add_job(q, puts, "5");
    tsq_add_job(q, puts, "6");

    size_t worker_count = 2;
    worker_pool p = wp_init(worker_count, q);

    while(!wp_is_empty(p)) {
        sleep(1);
    }
    wp_destroy(p);
    tsq_destroy(q);
```

Possible output:

```
1
3
2
5
6
4
```

Delegation

The workers are intended to be homogenous and interchangable. That way, any delegation of work can be split evenly across any available workers. If the workers are built in any kind of hierarchy, then this ability to delegate work will break down.

As such, almost every worker pool is a pool of equal peers. The difficulty arises in making sure that every member of the pool is actively being used by the application. This discussion is somewhat outside the realm of networking and would be best addressed in an Operating Systems class, dealing with concurrency and parallelism.

Chapter 10. Python/C API

The Python language is built on top of C code. While many Python libraries are in fact built with Python, some core part of the language must be built using a lower-level language, such as C.

The first step is setting up a simple build system. Thankfully, most of this is already included in Python and ready to go. Python includes a build system and C API that enable the programmer to create high-performance utilities (or existing libraries) with C, and port them to Python.

The second step is writing appropriate backing C code, although this could be written in any language capable of talking to a C API (such as C++ or Fortran).

There are a number of possible pitfalls in such an endeavor, but on the whole Python presents a fairly uniform interface at the low level; it merely becomes the job of the programmer to keep track of all the moving parts (just as in C programming).

Minimal Working Example

In Python, there is little setup required:

```
from distutils.core import setup, Extension

module_hello = Extension('hello', sources = ['hello.c'])

setup(name = 'Hello',
    version = '1.0',
    description = 'Hello package',
    ext_modules = [module_hello])
```

First, an extension unit is created. This is passed the name of the import (`'hello'`) as well as any C source files necessary to build the extension (`['hello.c']`).

This object, along with a few keywords, is passed to the `setup` function, which does the work of building the source files into libraries usable from Python. However, even with an empty `hello.c` file, not enough information is given to Python to build a module. The C file must also have some structure.

```c
#define PY_SSIZE_T_CLEAN
#include <Python.h>
// Python.h must come before all other headers

#include <stdlib.h>

// Need a PyMethodDef array of methods defined
static PyMethodDef methods[] = {
    {"method_name", func_ref, METH_VARARGS, "Method Docstring"},
    // Like a null-terminated string, Python uses null-terminated lists
    {NULL, NULL, 0, NULL}
};

// Need a PyModuleDef in static storage
static struct PyModuleDef module = {
    PyModuleDef_HEAD_INIT, "module_name",
    "Module docstring", -1, methods
};

// Match the name of the Python module
PyMODINIT_FUNC PyInit_hello(void)
{
    return PyModule_Create(&module);
}

static PyObject *func_ref(PyObject*self, PyObject*args)
{
    Py_RETURN_NONE;
}
```

All C libraries for Python must start with `#include <Python.h>` (or equivalent on a given system; version number may be required). This must come before any other normal C includes, `#define`s, or preprocessor instructions! This header file sets a number of options in specific ways and expects to be able to run first in order to do so.

The Python module needs three things:

1. The `PyModule_Create` function to be called on:
2. a `struct PyModuleDef` definition of the module, which needs:
3. a list of `PyMethodDef` method definitions exported by the module.

The `PyMethodDef` array is a `NULL`-terminated list of Python method descriptions. Each one is made up of the name of the method (as referred to in Python), the local C function that should be called (all of which have the same signature), the constant `METH_VARARGS`, followed by the docstring for the `help()` method in

Python.

While the constant `METH_VARARGS` could take a different value in older code, in practical terms that constant is the only value used today.

The `struct PyModuleDef` needs five values: The initializer macro (`PyModuleDef_HEAD_INIT`), the name of the module, its docstring, a size for each interpreter for the module (usually -1), and then a reference to the array of `PyMethodDef`s.

After these structures are built, the initialization function can be written. Its return type must be `PyMODINIT_FUNC`, which will set up an appropriate return type and some linkages required for the system (all other Python-interacting functions require a return type of `PyObject *`).

The name of this function **must** be of the form `PyInit`*`name`*`, where `*`name`* matches the on-disk name of the module used in `setup.py` when creating the `Extension` object. This function is run when the module is dynamically loaded by Python, so it is usual to have it call the `PyModule_Create` function, passing in a pointer to the earlier-defined `struct PyModuleDef` structure.

Methods

Exporting a method to Python requires an entry in the `PyMethodDef` array as described previously. But these methods require a very specific signature in order to be usable by Python:

```
PyObject *f(PyObject *, PyObject *);
```

Alternatively,

```
PyObject *f(PyObject*, PyObject*, PyObject*);
```

Note that every type in Python's C API is just a `PyObject *` by default. This reflects the underlying dynamic or run-time typing present in Python, where a single storage class of "variable" holds any type of object (Note carefully that this difference is distinct from the "strong vs. weak" typing axis!).

The first parameter is a `self` object, just like in Python. This will be the invoking object if this is a method, or the module in the case of a class function. The second parameter is the tuple of arguments to this call. An optional version takes three arguments, the third of which is a dictionary of keywords, if the `PyMethodDef` declaration stated that instead of the calling convention being `METH_VARARGS` it was listed as `METH_VARARGS|METH_KEYWORDS`.

Finally, all Python functions or methods must return a `PyObject *` value, which is allowed to be the Python `None` value, as in this example.

Building

To build the module, run `python3 setup.py build` in the same directory, and Python will build the appropriate C files and link them into a shared dynamic library, or `.so` file, under a `build` directory. This library can be loaded the same way as other Python code or libraries, via `import hello` with the `.so` in the same directory. From this point, `hello.method_name()` may be invoked.

```
$ cd build/lib*
$ python3
>>> import hello
>>> hello.method_name()
>>>
```

NULL vs Py_None

Every Python object is an object, including the `None` object. It has a class, methods, etc. As such, in the C API, `None` is treated as a kind of `PyObject *`. `NULL` is set aside to signal an exception being thrown in the function that returns `NULL` instead of a non-`NULL` `PyObject *`.

If a function returns `NULL` (rather than some other `PyObject *`, then that function is considered to have thrown an exception. The kind of exception is settable using `PyErr_*` family of functions, which set or test the current exception.

If a function detects that one of the functions **it** called fails, then it should generally not modify the exception, but let it percolate upward. This is just like writing exception-handling code in Python: rather than catch and re-throw exceptions, it is better to let an exception bubble up the stack, until a scope that is capable of dealing with that class of exception is reached.

Never store a `NULL` pointer into a Python object or something meant to be returned to the user of a Python program. The program is guaranteed to crash in an unfriendly way.

Reference Counting

Every object in Python uses a reference-counting mechanism to determine when to free the object in memory. As such, whenever objects are extracted from Python, their reference count must be incremented, and when they are no longer needed, the reference count must be decremented.

Confusingly, some Python functions automatically perform such operations, and getting the hang of which ones do and which ones do not can be difficult.

Any function that creates and returns a new object does increment its refcount. Any function that converts one object type to another increments the new object's refcount.

But, in general, an object passed to a function can be expected to borrow that object without incrementing its refcount. If the called function needs to keep track of the object for a longer period of time, then it should increment the refcount for the passed-in object before doing any further work.

Since a lot of functions can execute arbitrary Python code, leading to objects be garbage-collected at odd times, it is important to understand the basic rules regarding how refcounting works in Python.

New Python Types

It is possible to declare new `PyObject *` types that can be created and used from a compiled library using the C API.

```
typedef struct {
    PyObject_HEAD
    int gumball_count;
} gumball_machine_object;

static PyTypeObject gumball_machine_type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "gumball.Gumball",
    sizeof(gumball_machine_object),
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    Py_TPFLAGS_DEFAULT,
    "A gumball"
};
```

The first chunk of code describes what is stored for each object in memory when a reference to one is created. The second chunk of code sets up the type desired and any specific fields that it needs set. Note the large string of 0-values in the middle of this constructor; these are field unnecessary to this demonstration. Even more fields after that are also taken as zero.

The first line of both is a standard Python API macro, that expands into appropriate fields. The rest of the storage object contains any fields that need to be tracked on a per-instance basis, such as counts, file descriptors, and the like.

The object description (the second chunk of code) has quite a bit more involved. The second line is the type's "official name", the one that is printed in most error messages and documentation. The next parameter is just the size of each object (which may be different in heavy multiple inheritance; be careful).

Next come a series of the type's methods, which will not be examined in the demonstration, before arriving at `Py_TPFLAGS_DEFAULT`, which should be set thus for all modern Python code. Finally, a docstring may also be provided for the type.

Including the Class

To include this class in a module, it needs to be initialized and given a constructor. While a custom constructor is possible, this example will focus on a typical Python constructor.

```
// Match the name of the Python module
PyMODINIT_FUNC PyInit_hello(void)
{
    gumball_machine_type.tp_new = PyType_GenericNew;
    if(PyType_Ready(&gumball_machine_type) < 0) {
        return NULL;
    }

    PyObject *m = PyModule_Create(&module);
    if(!m) {
        return NULL;
    }

    PyINCREF(&gumball_machine_type);
    PyModule_AddObject(m, "Gumball", (PyObject *)&gumball_machine_type);
    return m;
}
```

The call `PyType_GenericNew` is a function that will do the normal object instantiation; no custom `__init__` or `__new__` functions involved. The `PyType_Ready` call does some initialization of the type; filling in fields that are blank with default values. Omitting this call will often yield unusual crashes.

The module is created as normal.

Since the type is an object (like all things in Python), it is manipulated as type `PyObject *` in the API. That means that its refcount must be incremented, since the module now owns a reference to it. This is done before adding the object to the module, and then returning the module.

This object, as created, has nothing unusual or interesting about it as present, other than a single field that is inaccessible from the Python side. Leveraging the C API beyond this point is outside the scope of this course, as it involves a drastic uptick in complexity.

Exercises

Exercise 1

Write a function using the C API that calculates the number of days in the year, when passed in a year.

Exercise 2

Write a function using the C API that calculates the number of days in a month when passed the month and the year as parameters.

Exercise 3

Write a function using the C API that returns the mean (average) of numbers in the collection passed in as a parameter.

Appendix A: Protocols

Ethernet

Ethernet is a data-level protocol designed to send traffic to one of many hosts all listening on the same bus. Each physical device has a relatively-unique ID assigned by the manufacturer know as a **MAC Address**.

Ethernet assumes that there is a single bus, or line of communication, shared between many hosts. When one host decides to send a message to a different host, it checks to see if the bus is already in use before attempting to transmit. If two hosts start transmitting at the same time, they are able to detect it, wait a random amount of time, and then try again.

Modern-day ethernet cables are **Full-Duplex**, with one pair of wires for sending and another pair for receiving. This has removed the risk of collisions, but the infrastructure to recover from them is still present.

Every machine on this shared bus listens for traffic that matches its MAC address. Such traffic is passed along to the operating system; other traffic is ignored. It is possible to put a network card into **promiscuous mode**, where it logs all traffic that it sees, not just the traffic intended for it.

Each message is wrapped in an Ethernet frame, a way of encapsulating the message for error detection. The size of the message may vary, from a minimum of 46 octets to a maximum of roughly 1500.

MAC Addresses

A MAC Address is a six-byte identifier for a (nominally) physical endpoint for Ethernet communications. The first three bytes identify the organization or manufacturer of the endpoint. The last three bytes work like a serial number, intended to be unique within that manufacturer (but not required). The combination of the two yields a *generally* globally unique signature. A MAC address is generally written as a series of hexadecimal bytes, separated by colons.

```
00:50:BA:31:7A:02
```

The MAC address may be changed in software (the operating system) to a different MAC address. This modification can be used to avoid a (very rare) collision in MACs, spoof a different machine, or evade MAC-restricted network policies.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Preamble																															
4	Preamble																								SOH Delimiter							
8	MAC Destination																															
12	MAC Dest.																Mac Src.															
16	MAC Source																															
20	Ethertype/Length																Payload															
24	Payload																															
28																																
32																																
36																																
40																																
44																																
48																																
52																																
56																																
60																																
64																																
68																																
⋮																																
N	Frame Check Sequence																															
N+4	Interpacket Gap																															
N+8																																

Point-to-Point Protocol (PPP)

The **Point-to-Point Protocol** is another data-layer protocol that is designed to be used when two nodes are talking on a dedicated line. Because of that assumption, the amount of overhead compared to Ethernet is incredibly small.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Flag								Address								Control								Protocol							
4	Protocol								Payload																							
⋮	Payload																															
N	Frame Check Sequence																Flag															

PPP is still a common protocol for its domain, such as dial-up Internet access, phone trunks, and other dedicated-line situations.

Address Resolution Protocol (ARP)

The **Address Resolution Protocol** acts as a bridge between the network and internet layers. It allows for discovery of a hardware address (MAC) for a given internet address (IP). In theory, this could be used across almost all network and internet protocols. In practice, it is used for Ethernet/IPv4. Ethernet/IPv6 uses the Neighbor Discovery Protocol.

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td colspan="16">Hardware Type</td><td colspan="16">Protocol Type</td></tr>
<tr><td>4</td><td colspan="8">Hardware Length</td><td colspan="8">Protocol Length</td><td colspan="16">Operation</td></tr>
<tr><td>8</td><td colspan="32">Sender Hardware Address</td></tr>
<tr><td>12</td><td colspan="16">Sender Hardware Address</td><td colspan="16">Sender Protocol Address</td></tr>
<tr><td>16</td><td colspan="16">Sender Protocol Address</td><td colspan="16">Target Hardware Address</td></tr>
<tr><td>20</td><td colspan="32">Target Hardware Address</td></tr>
<tr><td>24</td><td colspan="32">Target Protocol Address</td></tr>
</table>

IPv4

The **Internet Protocol, version 4 is** the modern-day Internet. It is an OSI network-layer protocol that is designed to route a message between two Internet hosts, regardless of how many other hosts are in between.

Each host is assigned an **IP Address**, a numeric label. Any host that participates in the Internet, upon receiving a packet addressed to a different IP address, tries to figure out the next host to send the payload along to.

If a host is close enough, the packet is sent directly. Otherwise, the packet is sent to a gateway that determines the next hop to make to get closer to the ultimate address. This path is not known at the time the packet is first sent! It is dynamically determined as the packet wends its way through the Internet.

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>4</td><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>8</td><td colspan="8">Time to Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>12</td><td colspan="32">Source IP Address</td></tr>
<tr><td>16</td><td colspan="32">Destination IP Address</td></tr>
</table>

Common Protocols

1	ICMP
6	TCP
17	UDP

IPv4 Addresses

IPv4 addresses are written as four integers separated by dots, such as 198.151.100.9. Each of these integers is the decimal version of the eight bits underlying that part of the 32-bit number that is the address. Unfortunately, their presentation in decimal hides the binary masking that is critical to understanding IPv4 addresses.

127.0.0.1 is the loopback address. This is a special address that always refers back to the current host machine.

It is useful to refer to a whole group of IP addresses at once. This is done using **network masks**, or netmasks. A netmask is written as an IP address followed by a number of fixed binary bits, and refers to all addresses that share that binary prefix.

IP	172								17								86								75							
IP_2	1	0	1	0	1	0	1	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	1	1	0	0	1	0	1	0	1	1
/12 mask	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Network	1	0	1	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Network	172								16								0								0							

A netmask of /12 would “fix” the twelve leading bits of the network address, allowing the other 20 bits to vary. This results in a total possible one million distinct addresses (1,048,576). A netmask of /32 would imply that all 32 bits are fixed, which would name exactly one address. The address 172.17.86.75 is included in the network 172.16.0.0/12.

Netmasks used to be specified in the same dotted notation as IP addresses, such as 255.240.0.0, but this has largely been replaced by the slash notation.

A few networks have special purposes or uses. The documentation networks are all class-C networks: 192.0.2.0/24, 198.51.100.0/24, and 203.0.113.0/24. The private networks that can be used by organizations are 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16.

IPv6

IPv4 allows for nearly four billion distinct hosts. However, that number has been exceeded due to nearly every device being connected to the Internet. This problem of running out of IP addresses was foreseen and addressed by the **Internet Protocol, version 6**.

Having nearly fifteen years of experience using IPv4, IPv6 was developed to streamline and expand the older protocol. A number of unecessary fields were removed, and the number of addresses was massively increased.

IPv6 allows for nearly 30 undecillion addresses; a 3 with 38 zeroes following. This excessive number of addresses means that it is unlikely that they will ever run out.

Other than the larger address space, IPv6 works in the same manner as IPv6. A few other changes are in place, such as the replacement of ARP with NDP and ICMPv4 with ICMPv6, but everything else in the network stack is untouched. This is a testament to the careful design of the original Internet protocol.

<table>
<tr><th></th><th colspan="4">0</th><th colspan="4">1</th><th colspan="4">2</th><th colspan="4">3</th><th colspan="4">4</th><th colspan="4">5</th><th colspan="4">6</th><th colspan="4">7</th><th colspan="4">8</th><th colspan="4">9</th><th colspan="4">10</th><th colspan="4">11</th><th colspan="4">12</th><th colspan="4">13</th><th colspan="4">14</th><th colspan="4">15</th><th colspan="4">16</th><th colspan="4">17</th><th colspan="4">18</th><th colspan="4">19</th><th colspan="4">20</th><th colspan="4">21</th><th colspan="4">22</th><th colspan="4">23</th><th colspan="4">24</th><th colspan="4">25</th><th colspan="4">26</th><th colspan="4">27</th><th colspan="4">28</th><th colspan="4">29</th><th colspan="4">30</th><th colspan="4">31</th></tr>
<tr><td>0</td><td colspan="16">Version</td><td colspan="16">Traffic Class</td><td colspan="32">Label</td></tr>
<tr><td>4</td><td colspan="48">Payload Length</td><td colspan="16">Next Header</td><td colspan="16">Hop Limit</td></tr>
<tr><td>8</td><td colspan="80" rowspan="4">Source IP Address</td></tr>
<tr><td>12</td></tr>
<tr><td>16</td></tr>
<tr><td>20</td></tr>
<tr><td>24</td><td colspan="80" rowspan="4">Destination IP Address</td></tr>
<tr><td>28</td></tr>
<tr><td>32</td></tr>
<tr><td>36</td></tr>
</table>

IPv6 Addresses

IPv6 addresses must be able to represent more than 30 undecillion distinct numbers (equivalent of 350 billion addresses per cubic centimeter of the earth). To do so, a few tricks are used.

To start with, all IPv6 addresses are viewed in hexadecimal. This allows for easy binary masking. Even so, this would mean that an address would be 32 hexadecimal digits long.

`2001:0db8:0000:0000:03e0:0000:18a7:8223` is an example of an IPv6 address. It is split up into eight two-byte chunks separated by colons.

Since this is still unwieldly, any leading 0s in a chunk may be omitted, leaving the simpler address of `2001:db8:0:0:3e0:0:18a7:8223`. Finally, the longest string of 0-only chunks can be abbreviated as ::, for a final address of `2001:db8::3e0:0:18a7:8223`. This :: abbreviation **only** happens to the longest string of zeros, nowhere else. If two strings of zeros have equal length, only the first group will use the abbreviation.

IPv6 networks can be specified with netmasks, just as IPv4 networks. The network `2001:db8::/32` would refer to a network of 4 billion addresses: the first 32 bits are fixed, and the remaining 32 are allowed to vary.

The local loopback address is `::1`. This is a special address that always refers back to the current host machine.

The documentation prefix, for examples, is `2001:db8::/32`. Hence the choice of network shown.

Private address spaces all live under the initial prefix of `fd00::/8`. The next 40 bits should be randomly generated to obtain a /48 network, which allows for 65,536 distinct hosts in the private network.

ICMP

ICMP is the **Internet Control Message Protocol**. It is designed for alerting Internet-capable hosts of failures or errors in delivering packets over the Internet.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Type								Code								Checksum															
8	Rest of Header																															

ICMPv6

For IPv6, the values of ICMP were altered to produce ICMPv6. The structure of the ICMPv6 header is the same as its version 4 counterpart.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Type								Code								Checksum															
8	Body																															

Neighbor Discovery Protocol (NDP)

The **Neighbor Discovery Protocol** is not a protocol at all, but a set of ICMPv6 messages.

UDP

The **User Datagram Protocol**, or UDP, is a simple protocol on top of IP that is designed for sending unreliable packets of data. Each packet, or datagram, is a fixed size, and there is no support for detecting whether or not the message was received successfully.

UDP is a fairly lightweight protocol, containing the bare minimum to deliver the datagram to its destination.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Source Port																Destination Port															
4	Length																Checksum															

TCP

The **Transport Control Protocol**, TCP, is known as a streaming protocol. Unlike UDP, it has features that allow for the detection and correction of missing packets, which allow for interactive communication between the two endpoints.

Under the hood, TCP is still transmitting datagrams, but they are assembled in-order by the receiving endpoint to give the appearance of a stream of bytes.

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td colspan="16">Source Port</td><td colspan="16">Destination Port</td></tr>
<tr><td>4</td><td colspan="32">Sequence Number</td></tr>
<tr><td>8</td><td colspan="32">Acknowlodgement Number</td></tr>
<tr><td>12</td><td colspan="4">Offset</td><td colspan="3">000</td><td>N</td><td>C</td><td>E</td><td>U</td><td>A</td><td>P</td><td>R</td><td>S</td><td>F</td><td colspan="16">Window Size</td></tr>
<tr><td>16</td><td colspan="16">Checksum</td><td colspan="16">Urgent Pointer</td></tr>
</table>

DNS

The **Domain Name System**, DNS, is a protocol that connects domain names like `example.org` with their IP addresses. There are two formats of message, queries and replies.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Identifier																QR	Opcode				AA	TC	RD	RA	000			Response		
4	Question Count																Answer Count															
8	NS Count																AR Count															