11 Networking

> The Web as I envisaged it, we have not seen it yet. The future is still so much bigger than the past
>
> Tim Berners-Lee

Networking has become arguably the most important use of computers in the past 10-20 years. Most of us nowadays can't stand a place without WiFi or any connectivity, so it is crucial as programmers that you have an understanding of networking and how to program to communicate across networks. Although it may sound complicated, POSIX has defined nice standards that make connecting to the outside world easy. POSIX also lets you peer underneath the hood and optimize all the little parts of each connection to write highly performant programs.

As an addendum that you'll read more about in the next chapter, we will be strict in our notation for sizes. That means that when we refer to the SI prefixes of Kilo-, Mega-, etc, then we are always referring to a power of 10. A kilobyte is one thousand bytes, a megabyte is a thousand kilobytes and so on. If we need to refer to 1024 bytes, we will use the more accurate term Kibibyte. Mibibyte and Gibibyte are the analogs of Megabyte and Gigabyte respectively. We make this distinction to make sure that we aren't off by 24. The reasons for this misnomer will be explained in the filesystems chapter.

The OSI Model

The Open Source Interconnection 7 layer model (OSI Model) is a sequence of segments that define standards for both infrastructure and protocols for forms of radio communication, in our case the Internet. The 7 layer model is as follows

1. Layer 1: The Physical Layer. These are the actual waves that carry the bauds across the wire. As an aside, bits don't cross the wire because in most mediums you can alter two characteristics of a wave – the amplitude and the frequency – and get more bits per clock cycle.
2. Layer 2: The Link Layer. This is how each of the agents reacts to certain events (error detection, noisy channels, etc). This is where Ethernet and WiFi live.
3. Layer 3: The Network Layer. This is the heart of the Internet. The bottom two protocols deal with communication between two different computers that are directly connected. This layer deals with routing packets from one endpoint to another.

4. Layer 4: The Transport Layer. This layer specifies how the slices of data are received. The bottom three layers make no guarantee about the order that packets are received and what happens when a packet is dropped. Using different protocols, this layer can.

5. Layer 5: The Session Layer. This layer makes sure that if a connection in the previous layers is dropped, a new connection in the lower layers can be established, and it looks like nothing happened to the end-user.

6. Layer 6: The Presentation Layer. This layer deals with encryption, compression, and data translation. For example, portability between different operating systems like translating newlines to windows newlines.

7. Layer 7: The Application Layer. HTTP and FTP are both defined at this level. This is typically where we define protocols across the Internet. As programmers, we only go lower when we think we can create algorithms that are more suited to our needs than all of the below.

This book won't cover networking in depth. We will focus on some aspects of layers 3, 4, and 7 because they are essential to know if you are going to be doing something with the Internet, which at some point in your career you will be. As for another definition, a protocol is a set of specifications put forward by the Internet Engineering Task Force that govern how implementers of a protocol have their program or circuit behave under specific circumstances.

Layer 3: The Internet Protocol

The following is a short introduction to internet protocol (IP), the primary way to send datagrams of information from one machine to another. "IP4", or more precisely, IPv4 is version 4 of the Internet Protocol that describes how to send packets of information across a network from one machine to another. Even as of 2018, IPv4 still dominates Internet traffic, but google reports that 24 countries now supply 15% of their traffic through IPv6 [2]. A significant limitation of IPv4 is that source and destination addresses are limited to 32 bits. IPv4 was designed at a time when the idea of 4 billion devices connected to the same network was unthinkable or at least not worth making the packet size larger. IPv4 addresses are written typically in a sequence of four octets delimited by periods "255.255.255.0" for example.

Each IPv4 datagram includes a small header - typically 20 octets, that includes a source and destination address. Conceptually the source and destination addresses can be split into two: a network number the upper bits and lower bits represent a particular host number on that network.

A newer packet protocol IPv6 solves many of the limitations of IPv4 like making routing tables simpler and 128-bit addresses. However, little web traffic is IPv6 based on comparison as of 2018 [2] We write IPv6 addresses in a sequence of eight, four hexadecimal delimiters like "1F45:0000:0000:0000:0000:0000:0000:0000". Since that can get unruly, we can omit the zeros "1F45::". A machine can have an IPv6 address and an IPv4 address.

There are special IP Addresses. One such in IPv4 is `127.0.0.1`, IPv6 as `0:0:0:0:0:0:0:1` or `::1` also known as localhost. Packets sent to 127.0.0.1 will never leave the machine; the address is specified to be the same machine. There are a lot of others that are denoted by certain octets being zeros or 255, the maximum value. You won't need to know all the terminology, keep in mind that the actual number of IP addresses that a machine can have globally over the Internet is smaller than the number of "raw" addresses. This book covers how IP deals with routing, fragmenting, and reassembling upper-level protocols. A more in-depth aside follows.

What's the deal with IPv6?

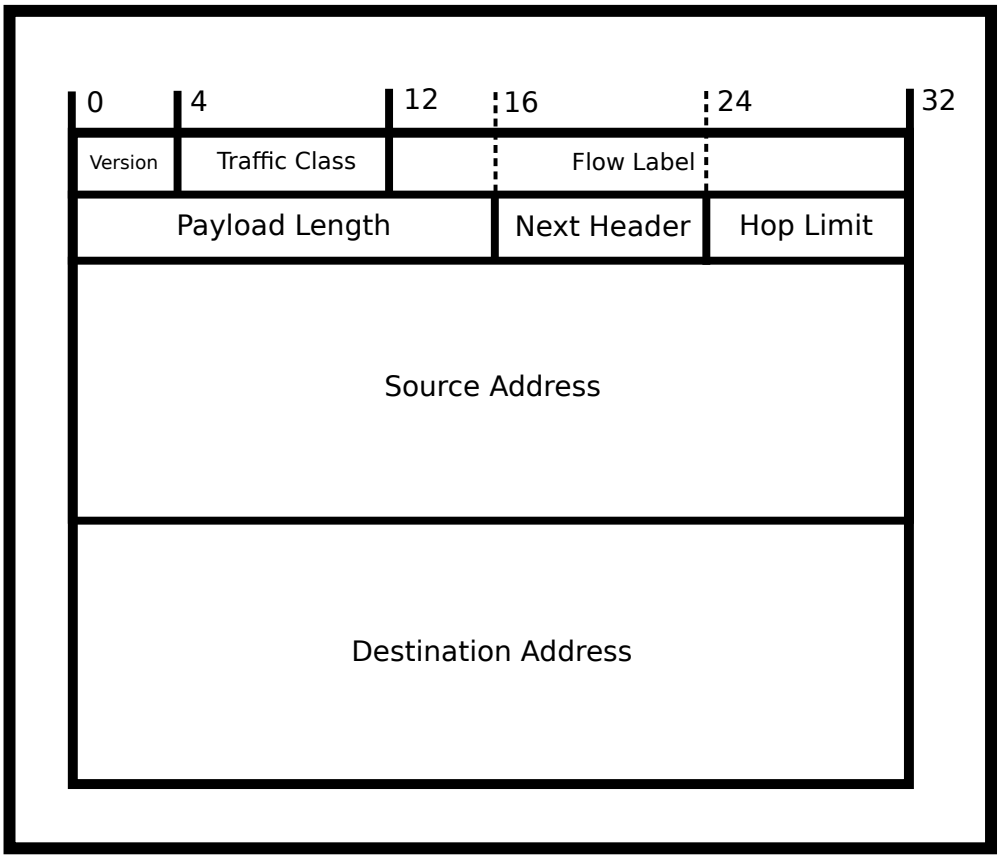

Figure 11.1: IPv6 Datagram divisibility

One of the big features of IPv6 is the address space. The world ran out of IP addresses a while ago and has been using hacks to get around that. With IPv6 there are enough internal and external addresses so even if we discover alien civilizations, we probably won't run out. The other benefit is that these addresses are leased not bought, meaning that if something drastic happens in let's say the Internet of things and there needs to be a change in the block addressing scheme, it can be done.

Another big feature is security through IPsec. IPv4 was designed with little to no security in mind. As such, now there is a key exchange similar to TLS in higher layers that allows you to encrypt communication.

Another feature is simplified processing. To make the Internet fast, IPv4 and IPv6 headers are verified in hardware. That means that all header options are processed in circuits as they come in. The problem is that as the IPv4 spec grew to include a copious amount of headers, the hardware had to become more and more advanced to support those headers. IPv6 reorders the headers so that packets can be dropped and routed with fewer hardware cycles. In the case of the Internet, every cycle matters when trying to route the world's traffic.

What's My Address?

To obtain a linked list of IP addresses of the current machine use `getifaddrs` which will return a linked list of IPv4 and IPv6 IP addresses among other interfaces as well. We can examine each entry and use `getnameinfo` to print the host's IP address. The `ifaddrs` struct includes the family but does not include the sizeof the struct. Therefore we need to manually determine the struct sized based on the family.

```
(family == AF_INET) ? sizeof(struct sockaddr_in) : sizeof(struct
    sockaddr_in6)
```

The complete code is shown below.

```
int required_family = AF_INET; // Change to AF_INET6 for IPv6
struct ifaddrs *myaddrs, *ifa;
getifaddrs(&myaddrs);
char host[256], port[256];

for (ifa = myaddrs; ifa != NULL; ifa = ifa->ifa_next) {
  int family = ifa->ifa_addr->sa_family;
  if (family == required_family && ifa->ifa_addr) {
    int ret = getnameinfo(ifa->ifa_addr,
    (family == AF_INET) ? sizeof(struct sockaddr_in) :
    sizeof(struct sockaddr_in6),
    host, sizeof(host), port, sizeof(port)
    , NI_NUMERICHOST | NI_NUMERICSERV)
    if (0 == ret) {
      puts(host);
    }
  }
}
```

To get your IP Address from the command line use `ifconfig` or Windows' `ipconfig`.
However, this command generates a lot of output for each interface, so we can filter the output using grep.

```
ifconfig | grep inet

Example output:
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
    inet6 fe80::7256:81ff:fe9a:9141%en1 prefixlen 64 scopeid 0x5
    inet 192.168.1.100 netmask 0xffffff00 broadcast 192.168.1.255
```

To grab the IP Address of a remote website, The function `getaddrinfo` can convert a human-readable domain name (e.g. `www.illinois.edu`) into an IPv4 and IPv6 address. It will return a linked-list of addrinfo structs:

```
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;
};
```

For example, suppose you wanted to find out the numeric IPv4 address of a web server at `www.bbc.com`. We do this in two stages. First, use getaddrinfo to build a linked-list of possible connections. Secondly, use `getnameinfo` to convert the binary address of one of those into a readable form.

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

struct addrinfo hints, *infoptr; // So no need to use memset
    global variables

int main() {
  hints.ai_family = AF_INET; // AF_INET means IPv4 only addresses

  // Get the machine addresses
  int result = getaddrinfo("www.bbc.com", NULL, &hints, &infoptr);
  if (result) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
    exit(1);
  }

  struct addrinfo *p;
  char host[256];

  for(p = infoptr; p != NULL; p = p->ai_next) {
    // Get the name for all returned addresses
    getnameinfo(p->ai_addr, p->ai_addrlen, host, sizeof(host),
        NULL, 0, NI_NUMERICHOST);
    puts(host);
```

```
  }

  freeaddrinfo(infoptr);
  return 0;
}
```

Possible output.

```
212.58.244.70
212.58.244.71
```

One can specify IPv4 or IPv6 with AF_UNSPEC. Just replace the ai_family attribute in the above code with the following.

```
hints.ai_family = AF_UNSPEC
```

If you are wondering how the computer maps hostnames to addresses, we will talk about that in Layer 7. Spoiler: It is a service called DNS. Before we move onto the next section, it is important to note that a single website can have multiple IP addresses. This may be to be efficient with machines. If Google or Facebook has a single server routing *all* of their incoming requests to other computers, they'd have to spend massive amounts of money on that computer or data center. Instead, they can give different regions different IP addresses and have a computer pick. It isn't bad to access a website through the non-preferred IP address. The page may load slower.

Layer 4: TCP and Client

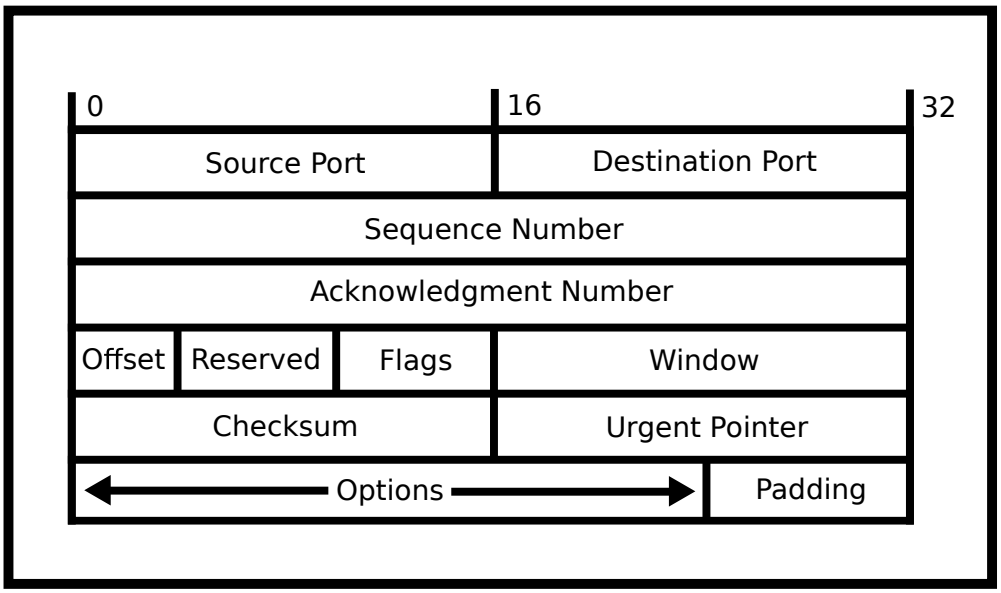

Figure 11.2: Extra: TCP Header Specification

Most services on the Internet today use TCP because it efficiently hides the complexity of the lower, packet-level nature of the Internet. TCP or Transport Control Protocol is a connection-based protocol that is built on top of IPv4 and IPv6 and therefore can be described as "TCP/IP" or "TCP over IP". TCP creates a *pipe* between two machines and abstracts away the low-level packet-nature of the Internet. Thus, under most conditions, bytes sent over a TCP connection delivered and uncorrupted. High performance and error-prone code won't even assume that!

TCP has many features that set it apart from the other transport protocol UDP.

1. Ports With IP, you are only allowed to send packets to a machine. If you want one machine to handle multiple flows of data, you have to do it manually with IP. TCP gives the programmer a set of virtual sockets. Clients specify the socket that you want the packet sent to and the TCP protocol makes sure that applications that are waiting for packets on that port receive that. A process can listen for incoming packets on a particular port. However, only processes with super-user (root) access can listen on ports less than 1024. Any process can listen on ports 1024 or higher. A frequently used port is number 80. It is used for unencrypted HTTP requests or web pages. For example, if a web browser connects to `http://www.bbc.com/` then it will be connecting to port 80.

2. Retransmission Packets can get dropped due to network errors or congestion. As such, they need to be retransmitted. At the same time, the retransmission shouldn't cause packets more packets to be dropped. This needs to balance the tradeoff between flooding the network and speed.

3. Out of order packets. Packets may get routed more favorably due to various reasons in IP. If a later packet arrives before another packet, the protocol should detect and reorder them.

4. Duplicate packets. Packets can arrive twice. Packets can arrive twice. As such, a protocol needs to be able to differentiate between two packets given a sequence number subject to overflow.

5. Error correction. There is a TCP checksum that handles bit errors. This is rarely used though.

6. Flow Control. Flow control is performed on the receiver side. This may be done so that a slow receiver doesn't get overwhelmed with packets. Servers that handle 10000 or 10 million concurrent connections may need to tell receivers to slow down but remain connected due to load. There is also the problem of making sure the local network's traffic is stable.

7. Congestion control. Congestion control is performed on the sender's side. Congestion control is to avoid a sender from flooding the network with too many packets. This is important to make sure that each TCP connection is treated fairly. Meaning that two connections leaving a computer to google and youtube receive the same bandwidth and ping as each other. One can easily define a protocol that takes all the bandwidth and leaves other protocols in the dust, but this tends to be malicious because many times limiting a computer to a single TCP connection will yield the same result.

8. Connection-Oriented/life cycle oriented. You can imagine a TCP connection as a series of bytes sent through a pipe. There is a "lifecycle" to a TCP connection though. TCP handles setting up the connection through SYN SYN-ACK ACK. This means the client will send a SYNchronization packet that tells TCP what starting sequence to start on. Then the receiver will send a SYN-ACK message acknowledging the synchronization number. Then the client will ACKnowledge that with one last packet. The connection is now open for both reading and writing on both ends TCP will send data and the receiver of the data will acknowledge that it received a packet. Then every so often if a packet is not sent, TCP will trade zero-length packets to make sure the connection is still alive. At any point, the client and server can send a FIN packet meaning that the server will not transmit. This packet can be altered with bits that only close the read or write end of a particular connection. When all ends are closed then the connection is over.

TCP doesn't provide many things, though.

1. Security. Connecting to an IP address claiming to be a certain website does not verify the claim (like in TLS). You could be sending packets to a malicious computer.

2. Encryption. Anybody can listen in on plain TCP. The packets in transport are in plain text. Important things like your passwords could easily be skimmed by onlookers.

3. Session Reconnection. If a TCP connection dies then a whole new one must be created, and the transmission has to be started over again. This is handled by a higher protocol.

4. Delimiting Requests. TCP is naturally connection-oriented. Applications that are communicating over TCP need to find a unique way of telling each other that this request or response is over. HTTP delimits the header through two carriage returns and uses either a length field or one keeps listening until the connection closes

Note on network orders

Integers can be represented in the least significant byte first or most significant byte first. Either approach is reasonable as long as the machine itself is internally consistent. For network communications, we need to standardize on the agreed format.

`htons(xyz)` returns the 16-bit unsigned integer 'short' value xyz in network byte order. `htonl(xyz)` returns the 32-bit unsigned integer 'long' value xyz in network byte order. Any longer integers need to have the computers specify the order.

These functions are read as 'host to network'. The inverse functions (`ntohs`, `ntohl`) convert network ordered byte values to host-ordered ordering. So, is host-ordering little-endian or big-endian? The answer is - it depends

on your machine! It depends on the actual architecture of the host running the code. If the architecture happens to be the same as network ordering then the functions return identical integers. For x86 machines, the host and network order are different.

Unless agreed otherwise, whenever you read or write the low-level C network structures, i.e. port and address information, remember to use the above functions to ensure correct conversion to/from a machine format. Otherwise, the displayed or specified value may be incorrect.

This doesn't apply to protocols that negotiate the endianness before-hand. If two computers are CPU bound by converting the messages between network orders – this happens with RPCs in high-performance systems – it may be worth it to negotiate if they are on similar endianness to send in little-endian order.

Why is network order defined to be big-endian? The simple answer is that RFC1700 says so [5]. If you want more information, we'll cite the famous article located that argued for a particular version [3]. The most important part is that it is standard. What happens when we don't have one standard? We have 4 different USB plug types (Regular, Micro, Mini, and USB-C) that don't interact well with each other. Include relevant XKCD here Standards.

TCP Client

There are three basic system calls to connect to a remote machine.

1. `int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct`

   The `getaddrinfo` call if successful, creates a linked-list of `addrinfo` structs and sets the given pointer to point to the first one.

   Also, you can use the hints struct to only grab certain entries like certain IP protocols, etc. The addrinfo structure that is passed into `getaddrinfo` to define the kind of connection you'd like. For example, to specify stream-based protocols over IPv6, you can use the following snippet.

```
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));

hints.ai_family = AF_INET6; // Only want IPv6 (use AF_INET for
    IPv4)
hints.ai_socktype = SOCK_STREAM; // Only want stream-based
    connection
```

   The other modes for 'family' are `AF_INET4` and `AF_UNSPEC` which mean IPv4 and unspecified respectively. This could be useful if you are searching for a service that you aren't entirely sure which IP version. Naturally, you get the version in the field back if you specified UNSPEC.

   Error handling with `getaddrinfo` is a little different. The return value *is* the error code. To convert to a human-readable error use `gai_strerror` to get the equivalent short English error text.

```
int result = getaddrinfo(...);
if(result) {
  const char *mesg = gai_strerror(result);
  ...
```

```
}
```

2. `int socket(int domain, int socket_type, int protocol);`

   The socket call creates a network socket and returns a descriptor that can be used with `read` and `write`. In this sense, it is the network analog of `open` that opens a file stream – except that we haven't connected the socket to anything yet!

   Sockets are created with a domain `AF_INET` for `IPv4` or `AF_INET6` for `IPv6`, `socket_type` is whether to use UDP, TCP, or other some other socket type, the `protocol` is an optional choice of protocol configuration for our examples this we can leave this as 0 for default. This call creates a socket object in the kernel with which one can communicate with the outside world/network. You can use the result of `getaddressinfo` to fill in the `socket` parameters, or provide them manually.

   The socket call returns an integer - a file descriptor - and, for TCP clients, you can use it as a regular file descriptor. You can use `read` and `write` to receive or send packets.

   TCP sockets are similar to `pipes` and are often used in situations that require IPC. We don't mention it in the previous chapters because it is overkill using a device suited for networks to simply communicate between processes on a single thread.

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

   Finally, the connect call attempts the connection to the remote machine. We pass the original socket descriptor and also the socket address information which is stored inside the addrinfo structure. There are different kinds of socket address structures that can require more memory. So in addition to passing the pointer, the size of the structure is also passed. To help identify errors and mistakes it is good practice to check the return value of all networking calls, including `connect`

```
// Pull out the socket address info from the addrinfo struct:
connect(sockfd, p->ai_addr, p->ai_addrlen)
```

4. (Optional) To clean up code call `freeaddrinfo(struct addrinfo *ai)` on the first level `addrinfo` struct.

There is an old function `gethostbyname` is deprecated. It's the old way convert a hostname into an IP address. The port address still needs to be manually set using `htons` function. It's much easier to write code to support IPv4 AND IPv6 using the newer `getaddrinfo`

This is all that is needed to create a *simple* TCP client. However, network communications offer many different levels of abstraction and several attributes and options that can be set at each level. For example, we haven't talked about `setsockopt` which can manipulate options for the socket. You can also mess around with lower protocols as the kernel provides primitives that contribute to this. Note that you need to be root to create a raw socket. Also, you need to have a lot of "set up" or starter code, be prepared to have your datagrams be dropped due to bad form as well. For more information see this guide.

Sending some data

Once we have a successful connection we can read or write like any old file descriptor. Keep in mind if you are connected to a website, you want to conform to the HTTP protocol specification to get any sort of meaningful results back. There are libraries to do this. Usually, you don't connect at the socket level. The number of bytes read or written may be smaller than expected. Thus, it is important to check the return value of read and `write`. A simple HTTP client that sends a request to a compliant URL is below. First, we'll start with the boring stuff and the parsing code.

```
typedef struct _host_info {
  char *hostname;
  char *port;
  char *resource;
} host_info;

host_info *get_info(char *uri) {
  // ... Parses the URI/URL
}

void free_info(host_info *info) {
  // ... Frees any info
}

int main(int argc, char *argv[]) {
  if(argc != 2) {
    fprintf(stderr, "Usage: %s http://hostname[:port]/path\n",
        *argv);
    return 1;
  }
  char *uri = argv[1];
  host_info *info = get_info(uri);
  host_info *temp = send_request(info);

  return 0;
}
```

The code that sends the request is below. The first thing that we have to do is connect to an address.

```
struct addrinfo current, *result;
memset(&current, 0, sizeof(struct addrinfo));
current.ai_family = AF_INET;
current.ai_socktype = SOCK_STREAM;

getaddrinfo(info->hostname, info->port, &current, &result);
```

```
connect(sock_fd, result->ai_addr, result->ai_addrlen)

freeaddrinfo(result);
```

The next piece of code sends the request. Here is what each header means.

1. "GET %s HTTP/1.0" This is the request verb interpolated with the path. This means to perform the GET verb on the path using the HTTP/1.0 method.
2. "Connection: close" Means that as soon as the request is over, please close the connection. This line won't be used for any other connections. This is a little redundant given that HTTP 1.0 doesn't allow you to send multiple requests, but it is better to be explicit given there are non-conformant technologies.
3. "Accept: */*" This means that the client is willing to accept anything.

A more robust piece of code would also check if the write fails or if the call was interrupted.

```
char *buffer;
asprintf(&buffer,
  "GET %s HTTP/1.0\r\n"
  "Connection: close\r\n"
  "Accept: */*\r\n\r\n",
  info->resource);

write(sock_fd, buffer, strlen(buffer));
free(buffer);
```

The last piece of code is the driver code that sends the request. Feel free to use the following code if you want to open the file descriptor as a FILE object for convenience functions. Just be careful not to forget to set the buffering to zero otherwise you may double buffer the input, which would lead to performance problems.

```
void send_request(host_info *info) {
  int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
  // Re-use address is a little overkill here because we are making
     a
  // Listen only server and we don't expect spoofed requests.
  int optval = 1;
  int retval = setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR,
     &optval,
  sizeof(optval));
  if(retval == -1) {
    perror("setsockopt");
    exit(1);
  }
  // Connect using code snippet
```

```
  // Send the get request

  // Open so you can use getline
  FILE *sock_file = fdopen(sock_fd, "r+");
  setvbuf(sock_file, NULL, _IONBF, 0);

  ret = handle_okay(sock_file);
  fclose(sock_file);
  close(sock_fd);
}
```

The example above demonstrates a request to the server using the HyperText Transfer Protocol. In general, there are six parts

1. The method. GET, POST, etc.
2. The resource. "/" "/index.html" "/image.png"
3. The protocol "HTTP/1.0"
4. A new line (`rn`). Requests always have a carriage return.
5. Any other knobs or switch parameters
6. The actual body of the request delimited by two new lines. The body of the request is either if the size is specified or until the receiver closes their connection.

The server's first response line describes the HTTP version used and whether the request is successful using a 3 digit response code.

```
HTTP/1.1 200 OK
```

If the client had requested a non-existent path, e.g. `GET /nosuchfile.html HTTP/1.0` Then the first line includes the response code is the well-known `404` response code.

```
HTTP/1.1 404 Not Found
```

For more information, RFC 7231 has the most current specifications on the most common HTTP method today [4].

Layer 4: TCP Server

The four system calls required to create a minimal TCP server are `socket`, `bind`, `listen`, and `accept`. Each has a specific purpose and should be called in roughly the above order

1. `int socket(int domain, int socket_type, int protocol)`

   To create an endpoint for networking communication. A new socket by itself is stores bytes. Though we've specified either a packet or stream-based connections, it is unbound to a particular network interface or port. Instead, socket returns a network descriptor that can be used with later calls to bind, listen and accept.

   As one gotcha, these sockets must be declared passive. Passive server sockets wait for another host to connect. Instead, they wait for incoming connections. Additionally, server sockets remain open when the peer disconnects. Instead, the client communicates with a separate active socket on the server that is specific to that connection.

   Since a TCP connection is defined by the sender address and port along with a receiver address and port, a particular server port there can be one passive server socket but multiple active sockets. One for each currently open connection. The server's operating system maintains a lookup table that associates a unique tuple with active sockets so that incoming packets can be correctly routed to the correct socket.

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

   The `bind` call associates an abstract socket with an actual network interface and port. It is possible to call bind on a TCP client. The port information used by bind can be set manually (many older IPv4-only C code examples do this), or be created using `getaddrinfo`.

   By default, a port is released after some time when the server socket is closed. Instead, the port enters a "TIMED-WAIT" state. This can lead to significant confusion during development because the timeout can make valid networking code appear to fail.

   To be able to immediately reuse a port, specify `SO_REUSEPORT` before binding to the port.

   ```
   int optval = 1;
   setsockopt(sfd, SOL_SOCKET, SO_REUSEPORT, &optval,
       sizeof(optval));

   bind(...);
   ```

   Here's an extended stackoverflow introductory discussion of `SO_REUSEPORT`.

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

   The `listen` call specifies the queue size for the number of incoming, unhandled connections. There are the connections unassigned to a file descriptor by `accept`. Typical values for a high-performance server are 128 or more.

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

   Once the server socket has been initialized the server calls `accept` to wait for new connections. Unlike `socket` `bind` and `listen`, this call will block, unless the nonblocking option has been set. If there are no new connections, this call will block and only return when a new client connects. The returned TCP socket is associated with a particular tuple `(client IP, client port, server IP, server port)` and will be used for all future incoming and outgoing TCP packets that match this tuple.

   Note the `accept` call returns a new file descriptor. This file descriptor is specific to a particular client. It is a common programming mistake to use the original server socket descriptor for the server I/O and then wonder why networking code has failed.

The accept system call can optionally provide information about the remote client, by passing in a sockaddr struct. Different protocols have different variants of the struct sockaddr, which are different sizes. The simplest struct to use is the sockaddr_storage which is sufficiently large to represent all possible types of sockaddr. Notice that C does not have any model of inheritance. Therefore we need to explicitly cast our struct to the 'base type' struct sockaddr.

```
struct sockaddr_storage clientaddr;
socklen_t clientaddrsize = sizeof(clientaddr);
int client_id = accept(passive_socket,
  (struct sockaddr *) &clientaddr,
  &clientaddrsize);
```

We've already seen getaddrinfo that can build a linked list of addrinfo entries and each one of these can include socket configuration data. What if we wanted to turn socket data into IP and port addresses? Enter getnameinfo that can be used to convert local or remote socket information into a domain name or numeric IP. Similarly, the port number can be represented as a service name. For example, port 80 is commonly used as the incoming connection port for incoming HTTP requests. In the example below, we request numeric versions for the client IP address and client port number.

```
socklen_t clientaddrsize = sizeof(clientaddr);
int client_id = accept(sock_id, (struct sockaddr *)
    &clientaddr, &clientaddrsize);
char host[NI_MAXHOST], port[NI_MAXSERV];
getnameinfo((struct sockaddr *) &clientaddr,
 clientaddrsize, host, sizeof(host), port, sizeof(port),
 NI_NUMERICHOST | NI_NUMERICSERV);
```

One can use the macros NI_MAXHOST to denote the maximum length of a hostname, and NI_MAXSERV to denote the maximum length of a port. NI_NUMERICHOST gets the hostname as a numeric IP address and similarly for NI_NUMERICSERV although the port is usually numeric, to begin with. The Open BSD man pages have more information

5. int close(int fd) and int shutdown(int fd, int how)

   Use the shutdown call when you no longer need to read any more data from the socket, write more data, or have finished doing both. When you call shutdown on socket on the read and/or write ends, that information is also sent to the other end of the connection. If you shut down the socket for further writing at the server end, then a moment later, a blocked read call could return 0 to indicate that no more bytes are expected. Similarly, a write to a TCP connection that has been shut down for reading will generate a SIGPIPE

   Use close when your process no longer needs the socket file descriptor.

   If you fork-ed after creating a socket file descriptor, all processes need to close the socket before the socket resources can be reused. If you shut down a socket for further read, all processes are affected because you've changed the socket, not the file descriptor. Well written code will shutdown a socket before calling close it.

There are a few gotchas to creating a server.

- Using the socket descriptor of the passive server socket (described above)
- Not specifying SOCK_STREAM requirement for getaddrinfo
- Not being able to reuse an existing port.
- Not initializing the unused struct entries
- The `bind` call will fail if the port is currently in use. Ports are per machine – not per process or user. In other words, you cannot use port 1234 while another process is using that port. Worse, ports are by default 'tied up' after a process has finished.

Example Server

A working simple server example is shown below. Note: this example is incomplete. For example, the socket file descriptor remains open and memory created by `getaddrinfo` remains allocated. First, we get the address info for our current machine.

```
struct addrinfo hints, *result;
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

int s = getaddrinfo(NULL, "1234", &hints, &result);
if (s != 0) {
  fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
  exit(1);
}
```

Then we set up the socket, bind it, and listen.

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

// Bind and listen
if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
  perror("bind()");
  exit(1);
}

if (listen(sock_fd, 10) != 0) {
  perror("listen()");
  exit(1);
}
```

We are finally ready to listen for connections, so we'll tell the user and accept our first client.

```
struct sockaddr_in *result_addr = (struct sockaddr_in *)
    result->ai_addr;
printf("Listening on file descriptor %d, port %d\n", sock_fd,
    ntohs(result_addr->sin_port));

// Waiting for connections like a passive socket
printf("Waiting for connection...\n");
int client_fd = accept(sock_fd, NULL, NULL);
printf("Connection made: client_fd=%d\n", client_fd);
```

After that, we can treat the new file descriptor as a stream of bytes much like a pipe.

```
char buffer[1000];
// Could get interrupted
int len = read(client_fd, buffer, sizeof(buffer) - 1);
buffer[len] = '\0';

printf("Read %d chars\n", len);
printf("===\n");
printf("%s\n", buffer);
```

[language=C]

Sorry To Interrupt

One concept that we need to make clear is that you need to handle interrupts in your networking code. That means that the sockets or accepted file descriptors that you read to or write to may have their calls interrupted – most of the time you will get an interrupt or two. In reality, any of your system calls could get interrupted. The reason we bring this up now is that you are usually waiting for the network. Which is an order of magnitude slower than processes. Meaning a higher probability of getting interrupted.

How would you handle interrupts? Let's try a quick example.

```
while bytes_read isn't count {
  bytes_read += read(fd, buf, count);
  if error is EINTR {
    continue;
  } else {
    break;
  }
}
```

We can assure you that the following code *experience errors*. Can you see why? On the surface, it does restart a call after a read or write. But what else happens when the error is EINTR? Are the contents of the buffer correct? What other problems can you spot?

Layer 4: UDP

UDP is a connectionless protocol that is built on top of IPv4 and IPv6. It's simple to use. Decide the destination address and port and send your data packet! However, the network makes no guarantee about whether the packets will arrive. Packets may be dropped if the network is congested. Packets may be duplicated or arrive out of order.

A typical use case for UDP is when receiving up to date data is more important than receiving all of the data. For example, a game may send continuous updates of player positions. A streaming video signal may send picture updates using UDP

UDP Attributes

- Unreliable Datagram Protocol Packets sent through UDP may be dropped on their way to the destination. This can especially be confusing because if you only test on your loop-back device – this is localhost or 127.0.0.1 for most users – then packets will seldom be lost because no network packets are sent.
- Simple The UDP protocol is supposed to have much less fluff than TCP. Meaning that for TCP there are a lot of configurable parameters and a lot of edge cases in the implementation. UDP is fire and forget.
- Stateless/Transaction The UDP protocol is stateless. This makes the protocol more simple and lets the protocol represent simple transactions like requesting or responding to queries. There is also less overhead to sending a UDP message because there is no three-way handshake.
- Manual Flow/Congestion Control You have to manually manage the flow and congestion control which is a double-edged sword. On one hand, you have full control over everything. On the other hand, TCP has *decades* of optimization, meaning your protocol for its use cases needs to be more efficient that to be more beneficial to use it.
- Multicast This is one thing that you can only do with UDP. This means that you can send a message to every peer connected to a particular router that is part of a particular group.

The full gory description is available at the original RFC [1].

While it may seem that you never want to use UDP for situations that you don't want to lose data, a lot of protocols base their communication based on UDP that requires complete data. Take a look at the Trivial File Transfer Protocol that reliably transmits a file over the wire using UDP only. Of course, there is more configuration involved, but choosing between UDP over TCP involves more than the above factors.

UDP Client

UDP Clients are pretty versatile below is a simple client that sends a packet to a server specified through the command line. Note that this client sends a packet and doesn't wait for an acknowledgment. It fires and forgets. The example below also uses gethostbyname because some legacy functionality still works pretty well for setting up a client.

```
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons((uint16_t)port);
struct hostent *serv = gethostbyname(hostname);
```

The previous code grabs an entry hostent that matches by hostname. Even though this isn't portable, it gets the job done. First is to connect to it and make it reusable – the same as a TCP socket. Note that we pass SOCK_DGRAM instead of SOCK_STREAM.

```
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
int optval = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEPORT, &optval,
    sizeof(optval));
```

Then, we can copy over our hostent struct into the sockaddr_in struct. Full definitions are provided in the man pages so it is safe to copy them over.

```
memcpy(&addr.sin_addr.s_addr, serv->h_addr, serv->h_length);
```

Then a final useful part of UDP is that we can time out receiving a packet as opposed to TCP because UDP isn't connection-oriented. The snippet to do that is below.

```
struct timeval tv;
tv.tv_sec = 0;
tv.tv_usec = SOCKET_TIMEOUT;
setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

Now, the socket is connected and ready to use. We can use sendto to send a packet. We should also check the return value. Note that we won't get an error if the packet isn't delivered because that is a part of the UDP protocol. We will, however, get error codes for invalid structs, bad addresses, etc.

```
```

```
char *to_send = "Hello!"
int send_ret = sendto(sock_fd, // Socket
   to_send, // Data
   strlen(to_send), // Length of data
   0, // Flags
   (struct sockaddr *)&ipaddr, // Address
   sizeof(ipaddr)); // How long the address is
```

The above code simply sends “Hello” through a UDP. There is no idea of if the packet arrives, is processed, etc.

UDP Server

There are a variety of function calls available to send UDP sockets. We will use the newer `getaddrinfo` to help set up a socket structure. Remember that UDP is a simple packet-based (‘datagram’) protocol. There is no connection to set up between the two hosts. First, initialize the hints addrinfo struct to request an IPv6, passive datagram socket.

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET6;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
```

Next, use getaddrinfo to specify the port number. We don’t need to specify a host as we are creating a server socket, not sending a packet to a remote host. Be careful not to send “localhost” or any other synonym for the loop-back address. We may end up trying to passively listen to ourselves and resulting in bind errors.

```
getaddrinfo(NULL, "300", &hints, &res);

sockfd = socket(res->ai_family, res->ai_socktype,
    res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
```

The port number is less than 1024, so the program will need `root` privileges. We could have also specified a service name instead of a numeric port value.

So far, the calls have been similar to a TCP server. For a stream-based service, we would call `listen` and accept. For our UDP-server, the program can start waiting for the arrival of a packet.

```
struct sockaddr_storage addr;
int addrlen = sizeof(addr);
```

```
// ssize_t recvfrom(int socket, void* buffer, size_t buflen, int
    flags, struct sockaddr *addr, socklen_t * address_len);

byte_count = recvfrom(sockfd, buf, sizeof(buf), 0, &addr,
    &addrlen);
```

The addr struct will hold the sender (source) information about the arriving packet. Note the `sockaddr_storage` type is sufficiently large enough to hold all possible types of socket addresses – IPv4, IPv6 or any other Internet Protocol. The full UDP server code is below.

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

int main(int argc, char **argv) {
  struct addrinfo hints, *res;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_INET6; // INET for IPv4
  hints.ai_socktype = SOCK_DGRAM;
  hints.ai_flags = AI_PASSIVE;

  getaddrinfo(NULL, "300", &hints, &res);

  int sockfd = socket(res->ai_family, res->ai_socktype,
      res->ai_protocol);

  if (bind(sockfd, res->ai_addr, res->ai_addrlen) != 0) {
    perror("bind()");
    exit(1);
  }
  struct sockaddr_storage addr;
  int addrlen = sizeof(addr);

  while(1){
    char buf[1024];
    ssize_t byte_count = recvfrom(sockfd, buf, sizeof(buf), 0,
        &addr, &addrlen);
    buf[byte_count] = '\0';

    printf("Read %d chars\n", byte_count);
    printf("===\n");
```

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

  return 0;
}
```

Note that if you perform a partial read from a packet, the rest of that data is discarded. One call to recvfrom is one packet. To make sure that you have enough space, use 64 KiB as storage space.

Layer 7: HTTP

Layer 7 of the OSI layer deals with application-level interfaces. Meaning that you can ignore everything below this layer and treat the Internet as a way of communicating with another computer than can be secure and the session may reconnect. Common layer 7 protocols are the following

1. HTTP(S) - Hypertext Transfer Protocol. Sends arbitrary data and executes remote actions on a web server. The S standards for secure where the TCP connection uses the TLS protocol to ensure that the communication can't be read easily by an onlooker.
2. FTP - File Transfer Protocol. Transfers a file from one computer to another
3. TFTP - Trivial File Transfer Protocol. Same as above but using UDP.
4. DNS - Domain Name Service. Translates hostnames to IP addresses
5. SMTP - Simple Mail Transfer Protocol. Allows one to send plain text emails to an email server
6. SSH - Secure SHell. Allows one computer to connect to another computer and execute commands remotely.
7. Bitcoin - Decentralized cryptocurrency
8. BitTorrent - Peer to peer file sharing protocol
9. NTP - Network Time Protocol. This protocol helps keep your computer's clock synced with the outside world

What's my name?

Remember when we were talking before about converting a website to an IP address? A system called "DNS" (Domain Name Service) is used. If the IP address is missing form a machine's cache then it sends a UDP packet to a local DNS server. This server may query other upstream DNS servers.

DNS by itself is fast but insecure. DNS requests are unencrypted and susceptible to 'man-in-the-middle' attacks. For example, a coffee shop internet connection could easily subvert your DNS requests and send back different IP addresses for a particular domain. The way this is usually subverted is that after the IP address is obtained then a connection is usually made over HTTPS. HTTPS uses what is called the TLS (formerly known as SSL) to secure transmissions and verify that the hostname is recognized by a Certificate Authority. Certificate Authorities often get hacked so be careful of equating a green lock to secure. Even with this added layer of security, the united

states government has recently issued a request for everyone to upgrade their DNS to DNSSec which includes additional security-focused technologies to verify with high probability that an IP address is truly associated with a hostname.

Digression aside, DNS works like this in a nutshell

1. Send a UDP packet to your DNS server
2. If that DNS server has the packet cached return the result
3. If not, ask higher-level DNS servers for the answer. Cache and send the result
4. If either packet is not answered from within a guessed timeout, resend the request.

If you want the full bits and pieces, feel free to look at the Wikipedia page. In essence, there is a hierarchy of DNS servers. First, there is the dot hierarchy. This hierarchy first resolves top-level domains `.edu` `.gov` etc. Next, it resolves the next level i.e. `illinois.edu`. Then the local resolvers can resolve any number of URLs. For example, the Illinois DNS server handles both `cs.illinois.edu` and `cs241.cs.illinois.edu`. There is a limit on how many subdomains you can have, but this is often used to route requests to different servers to avoid having to buy many high performant servers to route requests.

Non-Blocking IO

When you call `read()` if the data is unavailable, it will wait until the data is ready before the function returns. When you're reading data from a disk, that delay is short, but when you're reading from a slow network connection, requests take a long time. And the data may never arrive, leading to an unexpected close.

POSIX lets you set a flag on a file descriptor such that any call to `read()` on that file descriptor will return immediately, whether it has finished or not. With your file descriptor in this mode, your call to `read()` will start the read operation, and while it's working you can do other useful work. This is called "non-blocking" mode since the call to `read()` doesn't block.

To set a file descriptor to be non-blocking.

```
// fd is my file descriptor
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

For a socket, you can create it in non-blocking mode by adding `SOCK_NONBLOCK` to the second argument to `socket()`:

```
fd = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0);
```

When a file is in non-blocking mode and you call `read()`, it will return immediately with whatever bytes are available. Say 100 bytes have arrived from the server at the other end of your socket and you call `read(fd, buf, 150)`. 'read' will return immediately with a value of 100, meaning it read 100 of the 150 bytes you asked for. Say

you tried to read the remaining data with a call to `read(fd, buf+100, 50)`, but the last 50 bytes still hadn't arrived yet. `read()` would return -1 and set the global error variable **errno** to either `EAGAIN` or `EWOULDBLOCK`. That's the system's way of telling you the data isn't ready yet.

`write()` also works in non-blocking mode. Say you want to send 40,000 bytes to a remote server using a socket. The system can only send so many bytes at a time. In non-blocking mode, `write(fd, buf, 40000)` would return the number of bytes it was able to send immediately, or about 23,000. If you called `write()` right away again, it would return -1 and set errno to `EAGAIN` or `EWOULDBLOCK`. That's the system's way of telling you that it's still busy sending the last chunk of data and isn't ready to send more yet.

There are a few ways to check that your IO has arrived. Let's see how to do it using *select* and *epoll*. The first interface we have is select. It isn't preferred by many in the POSIX community if they have an alternative to it, and in most cases there is an alternative to it.

```
int select(int nfds,
fd_set *readfds,
fd_set *writefds,
fd_set *exceptfds,
struct timeval *timeout);
```

Given three sets of file descriptors, `select()` will wait for any of those file descriptors to become 'ready'.

1. `readfds` - a file descriptor in `readfds` is ready when there is data that can be read or EOF has been reached.
2. `writefds` - a file descriptor in `writefds` is ready when a call to write() will succeed.
3. `exceptfds` - system-specific, not well-defined. Just pass NULL for this.

`select()` returns the total number of ready file descriptors. If none of them become ready during the time defined by *timeout*, it will return 0. After `select()` returns, the caller will need to loop through the file descriptors in readfds and/or writefds to see which ones are ready. As readfds and writefds act as both input and output parameters, when `select()` indicates that there are ready file descriptors, it would have overwritten them to reflect only the ready file descriptors. Unless the caller intends to call `select()` only once, it would be a good idea to save a copy of readfds and writefds before calling it. Here is a comprehensive snippet.

```
fd_set readfds, writefds;
FD_ZERO(&readfds);
FD_ZERO(&writefds);
for (int i=0; i < read_fd_count; i++)
FD_SET(my_read_fds[i], &readfds);
for (int i=0; i < write_fd_count; i++)
FD_SET(my_write_fds[i], &writefds);

struct timeval timeout;
timeout.tv_sec = 3;
timeout.tv_usec = 0;
```

```
int num_ready = select(FD_SETSIZE, &readfds, &writefds, NULL,
    &timeout);

if (num_ready < 0) {
  perror("error in select()");
} else if (num_ready == 0) {
  printf("timeout\n");
} else {
  for (int i=0; i < read_fd_count; i++)
  if (FD_ISSET(my_read_fds[i], &readfds))
  printf("fd %d is ready for reading\n", my_read_fds[i]);
  for (int i=0; i < write_fd_count; i++)
  if (FD_ISSET(my_write_fds[i], &writefds))
  printf("fd %d is ready for writing\n", my_write_fds[i]);
}
```

For more information on select() The problem with select and why a lot of users don't use this or poll is that select must linearly go through each of the objects. If at any point in going through the objects, the previous objects change state, select must restart. This is highly inefficient if we have a large number of file descriptors in each of our sets. There is an alternative, that isn't much better.

epoll

epoll is not part of POSIX, but it is supported by Linux. It is a more efficient way to wait for many file descriptors. It will tell you exactly which descriptors are ready. It even gives you a way to store a small amount of data with each descriptor, like an array index or a pointer, making it easier to access your data associated with that descriptor.

First, you must create a special file descriptor with epoll_create(). You won't read or write to this file descriptor. You'll pass it to the other epoll_xxx functions and call close() on it at the end.

```
int epfd = epoll_create(1);
```

For each file descriptor that you want to monitor with epoll, you'll need to add it to the epoll data structures using epoll_ctl() with the EPOLL_CTL_ADD option. You can add any number of file descriptors to it.

```
struct epoll_event event;
event.events = EPOLLOUT; // EPOLLIN==read, EPOLLOUT==write
event.data.ptr = mypointer;
epoll_ctl(epfd, EPOLL_CTL_ADD, mypointer->fd, &event)
```

To wait for some of the file descriptors to become ready, use epoll_wait(). The epoll_event struct that it fills out will contain the data you provided in event.data when you added this file descriptor. This makes it easy for you to look up your data associated with this file descriptor.

```
int num_ready = epoll_wait(epfd, &event, 1, timeout_milliseconds);
if (num_ready > 0) {
  MyData *mypointer = (MyData*) event.data.ptr;
  printf("ready to write on %d\n", mypointer->fd);
}
```

Say you were waiting to write data to a file descriptor, but now you want to wait to read data from it. Just use `epoll_ctl()` with the `EPOLL_CTL_MOD` option to change the type of operation you're monitoring.

```
event.events = EPOLLOUT;
event.data.ptr = mypointer;
epoll_ctl(epfd, EPOLL_CTL_MOD, mypointer->fd, &event);
```

To unsubscribe one file descriptor from epoll while leaving others active, use `epoll_ctl()` with the `EPOLL_CTL_DEL` option.

```
epoll_ctl(epfd, EPOLL_CTL_DEL, mypointer->fd, NULL);
```

To shut down an epoll instance, close its file descriptor.

```
close(epfd);
```

Also to non-blocking `read()` and `write()`, any calls to `connect()` on a non-blocking socket will also be non-blocking. To wait for the connection to complete, use `select()` or epoll to wait for the socket to be writable. There are reasons to use epoll over select but due to interface, there are fundamental problems with doing so.

Blogpost about select being broken

Epoll Example

Let's break down the epoll code in the man page. We'll assume that we have a prepared TCP server socket `int listen_sock`. The first thing we have to do is create the epoll device.

```
epollfd = epoll_create1(0);
if (epollfd == -1) {
```

```
  perror("epoll_create1");
  exit(EXIT_FAILURE);
}
```

The next step is to add the listen socket in level triggered mode.

```
// This file object will be ‘read‘ from (connect is technically a
    read operation)
ev.events = EPOLLIN;
ev.data.fd = listen_sock;

// Add the socket in with all the other fds. Everything is a file
    descriptor
if (epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev) == -1) {
  perror("epoll_ctl: listen_sock");
  exit(EXIT_FAILURE);
}
```

Then in a loop, we wait and see if epoll has any events.

```
struct epoll_event ev, events[MAX_EVENTS];
nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
if (nfds == -1) {
  perror("epoll_wait");
  exit(EXIT_FAILURE);
}
```

If we get an event on a client socket, that means that the client has data ready to be read, and we perform that operation. Otherwise, we need to update our epoll structure with a new client.

```
if (events[n].data.fd == listen_sock) {
  int conn_sock = accept(listen_sock, (struct sockaddr *) &addr,
      &addrlen);
  // Must set to non-blocking
  setnonblocking(conn_sock);

  // We will read from this file, and we only want to return once
  // we have something to read from. We don't want to keep getting
  // reminded if there is still data left (edge triggered)
  ev.events = EPOLLIN | EPOLLET;
  ev.data.fd = conn_sock;
  epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock, &ev)
```

```
}
```

The function above is missing some error checking for brevity as well. Note that this code is performant because we added the server socket in level-triggered mode and we add each of the client file descriptors in edge-triggered. Edge triggered mode leaves more calculations on the part of the application – the application must keep reading or writing until the file descriptor is out of bytes – but it prevents starvation. A more efficient implementation would also add the listening socket in edge-triggered to clear out the backlog of connections as well.

Please read through most of `man 7 epoll` before starting to program. There are a lot of gotchas. Some of the more common ones will be detailed below.

Assorted Epoll Gotchas

There are several problems with using epoll. Here we will detail a few.

1. There are two modes. Level triggered and edge-triggered. Level triggered means that while the file descriptor has events on it, it will be returned by epoll when calling the ctl function. In edge-triggered, the caller will only get the file descriptor once it goes from zero events to an event. This means if you forget to read, write, accept etc on the file descriptor until you get a EWOULDBLOCK, that file descriptor will be dropped.
2. If at any point you duplicate a file descriptor and add it to epoll, you will get an event from that file descriptor and the duplicated one.
3. You can add an epoll object to epoll. Edge triggered and level-triggered modes are the same because ctl will reset the state to zero events
4. Depending on the conditions, you may get a file descriptor that was closed from Epoll. This isn't a bug. The reason that this happens is epoll works on the kernel object level, not the file descriptor level. If the kernel object lives longer and the right flags are set, a process could get a closed file descriptor. This also means that if you close the file descriptor, there is no way to remove the kernel object.
5. Epoll has the `EPOLLONESHOT` flag which will remove a file descriptor after it has been returned in `epoll_wait`
6. Epoll using level-triggered mode could starve certain file descriptors because it is unknown how much data the application will read from each descriptor.

Read more at `man 7 epoll` or check out a better version called `kqueue` in the appendix.

Remote Procedure Calls

RPC or Remote Procedure Call is the idea that we can execute a procedure on a different machine. In practice, the procedure may execute on the same machine. However, it may be in a different context. For example, the operation under a different user with different permissions and different lifecycles.

An example of this is you may send a remote procedure call to a docker daemon to change the state of the container. Not every application needs to have access to the entire system machine, but they should have access to containers that they've created.

Privilege Separation

The remote code will execute under a different user and with different privileges from the caller. In practice, the remote call may execute with more or fewer privileges than the caller. This in principle can be used to improve the security of a system by ensuring components operate with the least privilege. Unfortunately, security concerns need to be carefully assessed to ensure that RPC mechanisms cannot be subverted to perform unwanted actions. For example, an RPC implementation may implicitly trust any connected client to perform any action, rather than a subset of actions on a subset of the data.

Stub Code and Marshaling

The stub code is the necessary code to hide the complexity of performing a remote procedure call. One of the roles of the stub code is to *marshal* the necessary data into a format that can be sent as a byte stream to a remote server.

```
// On the outside, 'getHiscore' looks like a normal function call
// On the inside, the stub code performs all of the work to send
    and receive data to and from the remote machine.

int getHighScore(char* game) {
  // Marshal the request into a sequence of bytes:
  char* buffer;
  asprintf(&buffer,"getHiscore(%s)!", name);

  // Send down the wire (we do not send the zero byte; the '!'
      signifies the end of the message)
  write(fd, buffer, strlen(buffer) );

  // Wait for the server to send a response
  ssize_t bytesread = read(fd, buffer, sizeof(buffer));

  // Example: unmarshal the bytes received back from text into an
      int
  buffer[bytesread] = 0; // Turn the result into a C string

  int score= atoi(buffer);
  free(buffer);
  return score;
}
```

Using a string format may be a little inefficient. A good example of this marshaling is Golang's gRPC or Google RPC. There is a version in C as well if you want to check that out.

The server stub code will receive the request, unmarshal the request into a valid in-memory data call the underlying implementation and send the result back to the caller. Often the underlying library will do this for you.

To implement RPC you need to decide and document which conventions you will use to serialize the data into a byte sequence. Even a simple integer has several common choices.

1. Signed or unsigned?
2. ASCII, Unicode Text Format 8, some other encoding?
3. Fixed number of bytes or variable depending on the magnitude.
4. Little or Big endian binary format if using binary?

To marshal a struct, decide which fields need to be serialized. It may be unnecessary to send all data items. For example, some items may be irrelevant to the specific RPC or can be re-computed by the server from the other data items present.

To marshal a linked list, it is unnecessary to send the link pointers, stream the values. As part of unmarshaling, the server can recreate a linked list structure from the byte sequence.

By starting at the head node/vertex, a simple tree can be recursively visited to create a serialized version of the data. A cyclic graph will usually require additional memory to ensure that each edge and vertex is processed exactly once.

Interface Description Language

Writing stub code by hand is painful, tedious, error-prone, difficult to maintain and difficult to reverse engineer the wire protocol from the implemented code. A better approach is to specify the data objects, messages, and services to automatically generate the client and server code. A modern example of an Interface Description Language is Google's Protocol Buffer .proto files.

Even then, Remote Procedure Calls are significantly slower (10x to 100x) and more complex than local calls. An RPC must marshal data into a wire-compatible format. This may require multiple passes through the data structure, temporary memory allocation, and transformation of the data representation.

Robust RPC stub code must intelligently handle network failures and versioning. For example, a server may have to process requests from clients that are still running an early version of the stub code.

A secure RPC will need to implement additional security checks including authentication and authorization, validate data and encrypt communication between the client and host. A lot of the time, the RPC system can do this efficiently for you. Consider if you have both an RPC client and server on the same machine. Starting up a thrift or Google RPC server could validate and route the request to a local socket which wouldn't be sent over the network.

Transferring Structured Data

Let's examine three methods of transferring data using 3 different formats - JSON, XML, and Google Protocol Buffers. JSON and XML are text-based protocols. Examples of JSON and XML messages are below.

```
<ticket><price
    currency='dollar'>10</price><vendor>travelocity</vendor></ticket>
```

```
{ 'currency':'dollar' , 'vendor':'travelocity', 'price':'10' }
```

Google Protocol Buffers is an open-source efficient binary protocol that places a strong emphasis on high throughput with low CPU overhead and minimal memory copying. This means client and server stub code in multiple languages can be generated from the .proto specification file to marshal data to and from a binary stream.

Google Protocol Buffers reduces the versioning problem by ignoring unknown fields that are present in a message. See the introduction to Protocol Buffers for more information.

The general chain is to abstract away the actual business logic and the various marshaling code. If your application ever becomes CPU bound parsing XML, JSON or YAML, switch to protocol buffers!

Topics

- IPv4 vs IPv6
- TCP vs UDP
- Packet Loss/Connection Based
- Get address info
- DNS
- TCP client calls
- TCP server calls
- shutdown
- recvfrom
- epoll vs select
- RPC

Questions

- What is IPv4? IPv6? What are the differences between them?
- What is the TCP? The UDP? Give me the advantages and disadvantages of both of them. What is a scenario of using one over the other?
- Which protocol is connectionless and which one is connection based?
- What is DNS? What is the route that DNS takes?
- What does socket do?
- What are the calls to set up a TCP client?
- What are the calls to set up a TCP server?
- What is the difference between a socket shutdown and closing?
- When can you use `read` and `write`? How about `recvfrom` and `sendto`?

- What are some advantages to `epoll` over `select`? How about `select` over `epoll`?
- What is a remote procedure call? When should one use it versus HTTP or running code locally?
- What is marshaling/unmarshaling? Why is HTTP *not* an RPC?

Bibliography

[1] User Datagram Protocol. RFC 768, August 1980. URL `https://rfc-editor.org/rfc/rfc768.txt`.

[2] State of ipv6 deployment 2018, Jun 2018. URL `https://www.internetsociety.org/resources/2018/state-of-ipv6-deployment-2018/`.

[3] Danny Cohen. On holy wars and a plea for peace, Apr 1980. URL `https://www.ietf.org/rfc/ien/ien137.txt`.

[4] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014. URL `https://rfc-editor.org/rfc/rfc7231.txt`.

[5] J. Reynolds and J. Postel. Assigned numbers. RFC 1700, RFC Editor, October 1994.