Michael Whittaker

Beej's Guide to Network Programming

Thoughts and Notes

beejsockets github

Contents

1 Intro **5**
- 1.1 Audience . . . 5
- 1.2 Platform and Compiler . . . 5
- 1.3 Official Homepage and Books For Sale . . . 5
- 1.4 Note for Solaris/SunOS Programmers . . . 5
- 1.5 Note for Windows Programmers . . . 5

2 What is a socket? **6**
- 2.1 Two Types of Internet Sockets . . . 6
- 2.2 Low level Nonsense and Network Theory . . . 6

3 IP Adresses, structs, and Data Munging **8**
- 3.1 IP Addresses, versions 4 and 6 . . . 8
  - 3.1.1 Subnets . . . 8
  - 3.1.2 Port Numbers . . . 9
- 3.2 Byte Order . . . 9
- 3.3 structs . . . 9
- 3.4 IP Addresses, Part Deux . . . 11
  - 3.4.1 Private (Or Disconnected) Networks . . . 11

4 Jumping from IPv4 to IPv6 **12**

5 System Calls or Bust **13**
- 5.1 getaddrinfo() – Prepare to launch! . . . 13
- 5.2 socket() – Get the File Descriptor! . . . 15
- 5.3 bind() – What port am I on? . . . 16
- 5.4 connect() – Hey, you! . . . 17
- 5.5 listen() – Will somebody please call me? . . . 18
- 5.6 accept() – Thank you for calling port 3490 . . . 18
- 5.7 send() and recv() – Talk to me, baby! . . . 20
- 5.8 sendto() and recvfrom() – Talk to me, DGRAM-style . . . 20
- 5.9 close() and shutdown() – Get outta my face! . . . 21
- 5.10 getpeername() – Who are you? . . . 21
- 5.11 gethostname() – Who am I? . . . 21

6 Client-Server Background **22**

7 Slightly Advanced Techniques **23**

Listings

1 Identical IP Addresses . . . 8
2 addrinfo struct . . . 9
3 sockaddr struct . . . 10
4 sockaddr-in struct . . . 10
5 in-addr struct . . . 10
6 sockaddr-in6 struct . . . 10
7 in6-addr struct . . . 10
8 sockaddr-storage struct . . . 10
9 Presentation to Network . . . 11
10 Network to Presentation . . . 11
11 `getaddrinfo()` on local host port 3490 . . . 13
12 `getaddrinfo()` on google port 80 . . . 14
13 showip commanline utility . . . 14
14 Example socket use . . . 16
15 Example bind . . . 16
16 Example connect . . . 17
17 listen API . . . 18
18 Example accept . . . 19
19 send API . . . 20
20 recv API . . . 20
21 sendto API . . . 20
22 recvfrom API . . . 21
23 close API . . . 21
24 shutdown API . . . 21

Preface

This document contains the notes, musings, and thoughts generated during my reading of "Beej's Guide to Network Programming" by Brian Hall. The notes were taken primarily to encourage a thorough reading of the book and to help me recall the most important tidbits from the book upon a rereading of my notes. I can imagine the notes may be helpful to more than just me, so I am making them publicly available. A network programming novice, I cannot guarantee my notes are entirely correct, or even sensical at times. If you ever encounter a mistake, please contact me at mjw297@cornell.edu.

Along with the notes, I've thrown together some source code and other resources. Some of the code is taken directly from the text while some is original. All notes, code, and resources can be found at the beejsockets github.

Enjoy!

1 Intro

This book will teach network programming!

1.1 Audience

This is a tutorial, not a reference, for novice programmers.

1.2 Platform and Compiler

Compiled using Gnu's gcc.

1.3 Official Homepage and Books For Sale

Visit http://beej.us/guide/bgnet and http://beej.us/guide/url/bgbuy.

1.4 Note for Solaris/SunOS Programmers

You have to additional work (see the book).

1.5 Note for Windows Programmers

Switch to Unix :P

2 What is a socket?

A *socket* is a way to speak to other programs using standard Unix file descriptors. Recall that everything in Unix is a file. All I/O is done by reading and writing to a file descriptor, an integer associated with an open file. The file, however, can be many things: a network connection, a FIFO, a pipe, a terminal, etc. If we want to communicate with another program over the Internet, we'll do it via a file descriptor.We get, read, and write sockets using the `socket()`, `send()`, and `recv()` system calls.

There are many different kinds of sockets. This book will deal with DARPA Internet sockets.

2.1 Two Types of Internet Sockets

There are two types of sockets: "Stream Sockets" and "Datagram Sockets", also known as `SOCK_STREAM` and `SOCK_DGRAM` respectively.

Stream sockets Stream sockets are reliable two-way connected communication streams. The order of sent messages are maintained and the messaging is guaranteed to be error-free.

Applications such as telnet and the HTTP protocol use stream sockets.

Stream sockets use the Transmission Control Protocol, TCP, to guarantee their reliability.

Datagram sockets Datagram sockets are unreliable and connectionless. If you send a message it may not arrive and it may not arrive in the correct order. The only guarantee is that if the message does arrive, the data inside will be error-free.

Datagram sockets are connectionless because unlike stream sockets, you don't have to maintain an open connection. They are typically used in applications where dropping a few packets here and there is not important.

tftp, dhcpcd, multiplayer games, audio streaming, and video conferencing, all can use datagram sockets. Some applications like tftp and dhcpcd need additional protocols on top of UDP to ensure the packets make it, but other applications like gaming will simply ignore dropped packets (e.g. lag).

You would use and unreliable protocol like UDP for speed!

2.2 Low level Nonsense and Network Theory

Data encapsulation is how networking works. Essentially, data is wrapped in various headers and sent out, such as in Figure 1. When the packet is received, hardware will strip the ethernet header. The kernel will strip the IP and UDP headers. A TFTP program will stip the TFTP header and manipulate the unencapsulated data.

Such encapsulation is used in the *Layered Network Model.*

- Application

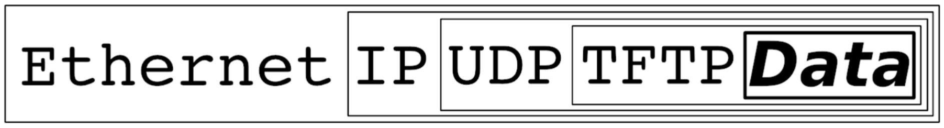

Figure 1: Data Encapsulation

- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

A model more consistent with Unix might be:

- Application Layer (telnet, ftp, etc.)
- Host-to-Host Transport Layer (TCP, UDP)
- Internet Layer (IP and routing)
- Network Access Layer (Ethernet, wi-fi, etc.)

3 IP Adresses, structs, and Data Munging

This section discusses IP addresses and ports as well as how the sockets API stores and manipulates IP addresses and other data.

3.1 IP Addresses, versions 4 and 6

Back when the Internet was originally created, we used IPv4. IP addresses were 32 bits (4 octets) and represented with "dots and numbers" as in `192.0.2.111`. However, as the number of required IP addresses grew, we ran out of IP addresses.

Enter IPv6. IPv6 addresses are 16 octets long and represented with colons and hexadecimal, as in `2001:odb8:c9d2:aee5:73e3:934a:a5ae:9551`. To compress IPv6 addresses, you can replace zeros with to colons. You can also leave off leading zeros in each byte pair. All of the pairs of IP addresses in Listing 1 are identical.

Listing 1: Identical IP Addresses

```
2001:0db8:c9d2:0012:0000:0000:0000:0051
2001:db8:c9d2:12::51

2001:0db8:ab00:0000:0000:0000:0000:0000
2001:db8:ab00::

0000:0000:0000:0000:0000:0000:0000:0001
::1
```

`::1` is the *loopback address* which is `127.0.0.1` in IPv4. IPv4 addresses can also be represented in IPv6. `192.0.2.33` translates to `::ffff:192.0.2.33`.

3.1.1 Subnets

For organizational purposes, it is convenient to label the first part of an IP address as the *network portion* of the address and the remaining part as the *host portion*. For example, consider the address `192.0.2.12`. The first three bytes could be the network and the last byte could be the host. That is, host `12` on network `192.0.2.0`.

In early versions of the Internet, there were different "classes" of subnets. Class A subnets had 1 byte of network. Class B subnets had 2 bytes of network. Class C subnets had 3 bytes of network. Eventually this scheme was deprecated and replaced with arbitrary length network portions.

The network portion of an address is described by a *netmask*, a set of bits you bitwise-AND the address with. For example, the netmask `255.255.255.0` yields three bytes of network. This scheme can also be expressed as an address followed by a forward slash followed by the number of bits in the network portion of the address. For example, `192.0.2.12/30` or `2001:db8::/32`.

3.1.2 Port Numbers

How do you multiplex different TCP or UDP applications on a computer with a single IP address? You use port numbers, a 16-bit number. Think of IP addresses as hotel addresses and port numbers as room numbers. Different applications run on different port numbers. HTTP runs on port 80, telnet on port 23, DOOM on port 666, etc.

3.2 Byte Order

Pretend you want to store the bytes `b34f` in your computer. Your computer can store them as `b3` then `4f`. This method, with the big end first, is known as *Big-Endian*. Other computers may store the bytes as `4f` then `b3` in a method known as *Little-Endian*.

Network Byte Order is synonymous with Big-Endian, and is the byte ordering sent across the network. *Host Byte Order* is the byte ordering of your computer. To convert to and from host and network ordering, we use 4 functions.

- `htons` host to network short
- `htonl` host to network long
- `ntohs` network to host short
- `ntohl` network to host long

3.3 structs

Refer to Listing 2, Listing 3, Listing 4, Listing 5, Listing 6, Listing 7, and Listing 8.

A socket descriptor is of type `int`.

A `addrinfo` struct is one of the first structs you'll interact with. It contains information about an address.

Listing 2: addrinfo struct

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next;      // linked list, next node
};
```

`sockaddr` contains a socket address. Dealing with the `sa_data` by hand is cumbersome. Instead, you can use `sockaddr_in` or `sockaddr_in6`. A pointer to a `sockaddr_in` can be cast to a pointer of `sockaddr` and vice-versa.

Listing 3: sockaddr struct

```c
struct sockaddr {
    unsigned short sa_family;   // address family, AF_xxx
    char           sa_data[14]; // 14 bytes of protocol address
};
```

Listing 4: sockaddr-in struct

```c
struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

Listing 5: in-addr struct

```c
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

Listing 6: sockaddr-in6 struct

```c
struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};
```

Listing 7: in6-addr struct

```c
struct in6_addr {
    unsigned char s6_addr[16]; // IPv6 address
};
```

`sockaddr_storage` is a struct large enough to hold both IPv4 and IPv6 structures.

Listing 8: sockaddr-storage struct

```c
struct sockaddr_storage {
    sa_family_t ss_family; // address family

    // all of this is padding, implementation specific, ignore it:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

3.4 IP Addresses, Part Deux

Fortunately, there are many functions to help manipulate IP addresses.

If you want to convert a string representation of an IP address into a representation suitable for a struct, use `inet_pton()`. `pton` stands for "presentation to network" or "printable to network". An example use is given in Listing 9. `inet_pton` returns -1 on error and 0 if the address is messed up.

Listing 9: Presentation to Network

```
struct sockaddr_in  sa;  // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr));
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));
```

The opposite conversion can be made using `inet_ntop()`, as shown in Listing 10.

Listing 10: Network to Presentation

```
char ip4[INET_ADDRSTRLEN];
struct sockaddr_in sa;
inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);
printf("The IPv4 address is: %s\n", ip4);

char ip6[INET6_ADDRSTRLEN];
struct sockaddr_in6 sa6;
inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);
printf("The address is: %s\n", ip6);
```

3.4.1 Private (Or Disconnected) Networks

Many networks are hidden behind a firewall that translates internal IP addresses to external IP addresses. This is done via a process known as NAT, or *Network Address Translation*.

Your public IP may be `192.0.2.33`, but your computer will say `10.x.x.x` or `192.168.x.x`.

4 Jumping from IPv4 to IPv6

A brief list of methods to switch from IPv4 to IPv6.

1. Use `getaddrinfo()`.
2. Wrap all references to IP version in helper functions.
3. Change `AF_INET` to `AF_INET6`.
4. Change `PF_INET` to `PF_INET6`.
5. Change `INADDR_ANY` to `in6addr_any`.
6. Use `sockaddr_in6` instead of `sockaddr_in`.
7. Use `in6_addr` instead of `in_addr`.
8. Use `inet_pton()` not `inet_aton()` or `inet_addr()`.
9. Use `inet_ntop()` not `inet_ntoa()`.
10. Use `getaddrinfo()` not `gethostbyname()`.
11. Use `getnameinfo()` not `gethostbyaddr()`.
12. Don't use `INADDR_BROADCAST`

5 System Calls or Bust

System calls allow us to access network functionality. When we invoke these functions, the kernel takes over and performs the service. The system calls are outlined in roughly the same order in which we will need to call them.

5.1 getaddrinfo() – Prepare to launch!

This function helps set up `struct`s that we will use later on. Listing 11 shows the code we would write if we are a server and want to listen on our host's IP address, port 3490. The code doesn't actually do any listening or network setup, but it does setup the structures we'll need later.

`AI_PASSIVE` tells `getaddrinfo()` to assign the address of my local host to the socket structures.

Listing 11: `getaddrinfo()` on local host port 3490

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

/*
int getaddrinfo(
    const char *node,             // e.g. "www.example.com" or IP
    const char *service,          // e.g. "http" or port number
    const struct addrinfo *hints,
    struct addrinfo **res
);
*/

int main() {
    struct addrinfo hints;
    struct addrinfo *servinfo;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;   // don't care IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
    hints.ai_flags    = AI_PASSIVE;  // fill in my IP for me

    int status = getaddrinfo(NULL, "3490", &hints, &servinfo);
    if (status != 0) {
        std::cerr << "getaddrinfo error: " << gai_strerror(status) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    std::cout << "success!" << std::endl;

    freeaddrinfo(servinfo);
}
```

If we are a client wanting to connect to google.com, we would use Listing 12.

Listing 12: `getaddrinfo()` on google port 80

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

int main() {
    struct addrinfo hints;
    struct addrinfo *servinfo;

    std::memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;   // don't care IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM; // TCP stream sockets

    int status = getaddrinfo("www.google.com", "80", &hints, &servinfo);
    if (status != 0) {
        std::cerr << "getaddrinfo error: " << gai_strerror(status) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    std::cout << "success!" << std::endl;

    freeaddrinfo(servinfo);
}
```

A more advanced example of `getaddrinfo` is shown in Listing 13.

Listing 13: showip commanline utility

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>

void usage();
void showip(std::string hostname);

void usage() {
    std::cout << "usage: showip <hostname>" << std::endl;
}

void showip(std::string hostname) {
    struct addrinfo hints;
    struct addrinfo *res;
    char ipstr[INET6_ADDRSTRLEN];

    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    int status = getaddrinfo(hostname.c_str(), NULL, &hints, &res);
    if (status != 0) {
        std::cerr << "getaddrinfo: " << gai_strerror(status) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    std::cout << "IP addresses for " << hostname << std::endl;

    for (struct addrinfo *p = res; NULL != p; p = p->ai_next) {
        if (AF_INET == p->ai_family) { // IPv4
            struct sockaddr_in *ipv4 = (struct sockaddr_in *) p->ai_addr;
            struct in_addr     *addr = &(ipv4->sin_addr);
            inet_ntop(p->ai_family, addr, ipstr, sizeof(ipstr));
            std::cout << "IPv4: " << ipstr << std::endl;
        }
        else { // IPv6
            struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *) p->ai_addr;
            struct in6_addr     *addr = &(ipv6->sin6_addr);
            inet_ntop(p->ai_family, addr, ipstr, sizeof(ipstr));
            std::cout << "IPv6: " << ipstr << std::endl;
        }
    }

    freeaddrinfo(res);
}

int main(int argc, char **argv) {
    if (argc < 2) {
        usage();
        std::exit(EXIT_FAILURE);
    }

    for (int i = 1; i < argc; ++i) {
        showip(argv[i]);
        std::cout << ((argc - 1) == i ? "" : "\n");
    }
}
```

5.2 socket() – Get the File Descriptor!

The `socket()` system call takes three arguments: domain, type, protocol. domain is `PF_INET` or `PF_INET6`. type is `SOCK_STREAM` or `SOCK_DGRAM` and protocol is 0 to choose the proper protocol from the type.

You used to pack these by hand, which you can still do, but it's better to use the structs formed from `getaddrinfo`, as shown in Listing 14. `socket()` returns a *socket descriptor* or

-1 on error. It also sets `errno` on error.

Listing 14: Example socket use

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

/*
int socket(int domain, int type, int protocol)
*/

int main() {
    struct addrinfo hints;
    struct addrinfo *res;

    std::memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    int status = getaddrinfo("www.google.com", "80", &hints, &res);
    if (status != 0) {
        std::cerr << "getaddrinfo error: " << gai_strerror(status) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    int s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    std::cout << s << std::endl;

    freeaddrinfo(res);
}
```

5.3 bind() – What port am I on?

Once you have a socket, you might have to associate it with a certain port; for example, if you are going to `listen()`. If you're a client and are only going to `connect()`, you may not need to `bind()`. Also make sure not to bind to any ports under 1024. An example bind is given in Listing **??**.

Listing 15: Example bind

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
```

```

/*
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
*/

int main() {
    struct addrinfo hints;
    struct addrinfo *res;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;   // don’t care IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
    hints.ai_flags    = AI_PASSIVE;  // fill in my IP for me

    int status = getaddrinfo(NULL, "3490", &hints, &res);
    if (status != 0) {
        std::cerr << "getaddrinfo error: " << gai_strerror(status) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    int s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (s == -1) {
        perror("");
        std::exit(EXIT_FAILURE);
    }

    int err = bind(s, res->ai_addr, res->ai_addrlen);
    if (err == -1) {
        perror("");
        std::exit(EXIT_FAILURE);
    }

    freeaddrinfo(res);
}
```

5.4 connect() – Hey, you!

Connecting is similar to binding and shown in Listing ??. Notice that we didn’t bind because we only care about the server’s port number.

Listing 16: Example connect

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

/*
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

```
*/

int main() {
    struct addrinfo hints;
    struct addrinfo *res;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;   // don't care IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM; // TCP stream sockets

    int status = getaddrinfo("www.google.com", "80", &hints, &res);
    if (status != 0) {
        std::cerr << "getaddrinfo error: " << gai_strerror(status) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    int s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (s == -1) {
        perror("");
        std::exit(EXIT_FAILURE);
    }

    int err = connect(s, res->ai_addr, res->ai_addrlen);
    if (err == -1) {
        perror("");
        std::exit(EXIT_FAILURE);
    }

    freeaddrinfo(res);
}
```

5.5 listen() – Will somebody please call me?

If we are a server and want to wait for incoming connections and handle them in some way, we will need to `listen()` and then `accept()`. We can `listen()` as shown in Listing 17

Listing 17: listen API

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

`sockfd` is the usual socket file descriptor. `backlog` is the number of connections allowed on the incoming queue. When other nodes connect to our port, they are arranged in a queue until we `accept()` them. `backlog` tells us how many we allow on the queue at any one time. We can set this value usually anywhere from 5-20.

An example of `listen()` is deferred to the section on `accept()`.

5.6 accept() – Thank you for calling port 3490

When other nodes connect to our port, they are queued as we listen. We must `accept` the nodes to handle them. `accept` returns a new socket file descriptor that is ready to `send()`

and `recv()`. An example accept is shown in Listing 18.

Listing 18: Example accept

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

/*
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
*/

namespace {
    const char *PORT    = "3490";
    const int  BACKLOG = 10;
}


int main() {
    struct sockaddr_storage their_addr;
    socklen_t addr_size = sizeof(sockaddr_storage);
    struct addrinfo hints;
    struct addrinfo *res;
    int sockfd;
    int new_fd;
    int err;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;   // don't care IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
    hints.ai_flags    = AI_PASSIVE;  // fill in my IP for me

    int status = getaddrinfo(NULL, PORT, &hints, &res);
    if (status != 0) {
        std::cerr << "getaddrinfo error: " << gai_strerror(status) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (sockfd == -1) {
        perror("");
        std::exit(EXIT_FAILURE);
    }

    err = bind(sockfd, res->ai_addr, res->ai_addrlen);
    if (err == -1) {
        perror("");
        std::exit(EXIT_FAILURE);
    }

    err = listen(sockfd, BACKLOG);
    if (err == -1) {
        perror("");
        std::exit(EXIT_FAILURE);
    }

    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);
    std::cout << new_fd << std::endl;

    freeaddrinfo(res);
}
```

5.7 send() and recv() – Talk to me, baby!

These two functions are used to communicate over stream sockets or connected datagram sockets.

The `send()` API is given in Listing 19. `sockfd` is the socket descriptor you want to send data to. `msg` is a pointer to the data we want to send. `len` is the length od the data. `flags` can be set to 0 for now.

`send()` returns the actual number of bytes sent. It is our responsibility to send any unsent data. -1 is returned and `errno` is set on error.

Listing 19: send API

```
int send(int sockfd, const void *msg, int len, int flags);
```

The `recv()` API is given in Listing 20. `sockfd` is the socket descriptor to read from. `buf` is the buffer to read the information into. `len` is the maximum length of the buffer. `flags` can be set to 0.

`recv()` returns the actual number of bytes read into the buffer. `recv()` returns 0 if the remote side has closed a connection. -1 is returned and `errno` is set upon error.

Listing 20: recv API

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

An example of send and receive is delayed until the chapter on the client-server model.

5.8 sendto() and recvfrom() – Talk to me, DGRAM-style

If you want to send data across unconnected datagram sockets, you need to use `sendto()` and `recvfrom()`. Their API is similar to `send()` and `recv()` and is shown in Listing 21 and Listing 22.

Listing 21: sendto API

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
        const struct sockaddr *to, socklen_t tolen);
```

Listing 22: recvfrom API

```c
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
        struct sockaddr *from, int *fromlen);
```

5.9 close() and shutdown() – Get outta my face!

If you want to close a file, simply call `close()`, as shown in Listing 23. If you want more control over how the socket is shut down, you can use `shutdown()`, as shown in Listing 24. `how` is one of the following.

- `0` Further receives are disallowed
- `1` Further sends are disallowed
- `2` Further sends and receives are disallowed

Note that `shutdown()` doesn't actually close the file descriptor. For that, you still need to use `close()`.

Listing 23: close API

```c
close(sockfd);
```

Listing 24: shutdown API

```c
int shutdown(int sockfd, int how);
```

5.10 getpeername() – Who are you?

`getpeername()` will tell you who is at the other end of a connected stream socket.

5.11 gethostname() – Who am I?

`gethostname()` gets the name of the computer being run on.

6 Client-Server Background

7 Slightly Advanced Techniques