

Networking, Part 4: Building a simple TCP Server

Alex Kizer edited this page 26 days ago · 2 revisions

What is `htons` and when is it used?

Integers can be represented in 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 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.

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 result of these functions is just the argument. For x86 machines, the host and network ordering *is* different.

Summary: Whenever you read or write the low level C network structures (e.g. 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.

What are the 'big 4' network calls used to create a server?

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

The port information (used by `bind`) can be set manually (many older IPv4-only C code examples do this), or be created using `getaddrinfo`

We also see examples of `setsockopt` later too.

What is the purpose of calling `socket` ?

To create a endpoint for networking communication. A new socket by itself is not particularly useful; though we've specified either a packet or stream-based connections it is not bound 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`.

What is the purpose of calling `bind`

Edit

New Page

▼ Pages 51

[Home](#)

[#Example Markdown](#)

[#Informal Glossary](#)

[#Piazza: When And How to Ask For Help](#)

[C Programming, Part 1: Introduction](#)

[C Programming, Part 2: Text Input And Output](#)

[C Programming, Part 3: Common Gotchas](#)

[C Programming, Part 4: Debugging](#)

[Deadlock, Part 1: Resource Allocation Graph](#)

[Deadlock, Part 2: Deadlock Conditions](#)

[File System, Part 1: Introduction](#)

[File System, Part 2: Files are inodes \(everything else is just data...\)](#)


[File System, Part 3: Permissions](#)

[File System, Part 4: Working with directories](#)


[File System, Part 5: Virtual file systems](#)


Show 36 more pages...


Clone this wiki locally


 Clone in Desktop

<>









The `bind` call associates an abstract socket with an actual network interface and port. It is possible to call `bind` on a TCP client however it's unusually unnecessary to specify the outgoing port.

What is the purpose of calling `listen`

The `listen` call specifies the queue size for the number of incoming, unhandled connections i.e. that have not yet been assigned a network descriptor by `accept`. Typical values for a high performance server are 128 or more.

Why are server sockets passive?

Server sockets do not actively try to connect to another host; instead they wait for incoming connections. Additionally, server sockets are not closed when the peer disconnects. Instead when a remote client connects, it is immediately bumped to an unused port number for future communications.

What is the purpose of calling `accept`

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. i.e. if there are no new connections, this call will block and only return when a new client connects.

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

What are the gotchas of creating a TCP-server?

- Using the socket descriptor of the passive server socket (described above)
- Not specifying `SOCK_STREAM` requirement for `getaddrinfo`
- Not being able to re-use an existing port.
- Not initializing the unused struct entries
- The `bind` call will fail if the port is currently in use

Note, ports are per machine- not per process or per 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.

Server code example

A working simple server example is shown below. Note this example is incomplete - for example it does not close either socket descriptor, or free up memory created by `getaddrinfo`

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

    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;

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

    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);
    }

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

    printf("Waiting for connection...\n");
    int client_fd = accept(sock_fd, NULL, NULL);
    printf("Connection made: client_fd=%d\n", client_fd);

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

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

    return 0;
}

```

Why can't my server re-use the port?

By default a port is not immediately released when the 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 re-use 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` .

Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

