Listen for and accept TCP connections in C

Content

Tested on

Debian (Lenny)
Ubuntu (Trusty)

Objective

To listen for and accept inbound TCP connections in C

Scenario

Suppose that you wish to write a daemon that implements the TCP-based variant of the Daytime Protocol, as defined by RFC 867

This is a very simple protocol whereby the server sends a human-readable copy of the current date and time then closes the connection. Any data that the client might send is ignored.

Method

Overview

The method described here has six steps:

1. Construct the local socket address.
2. Create the server socket.
3. Set the `SO_REUSEADDR` socket option.
4. Bind the local address to the server socket.
5. Listen for inbound connections.
6. Accept connections as they arrive.

The following header files will be needed:

```
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

Construct the local socket address

In order to listen for TCP connections it is necessary to choose a port number and, optionally, a local IP address on which to listen. The combination of these two values is treated as a single entity called the socket address, which is represented by a `struct sockaddr_in` for IPv4 or a `struct sockaddr_in6` for IPv6.

Most common network services have an assigned port number on which they are normally expected to listen. While it makes sense to use this as the default, it is good practice to make the port number configurable. Possible reasons for wanting to override the assigned port number include:

- running multiple instances of a network service on the same machine,
- running a network service that would normally use a well-known port number from a non-root account, or
- making port scanning more time-consuming than it would be if the standard port number were used.

The local IP address should normally default to either the the wildcard address or the loopback address, but like the port number it is good practice to make it configurable. When a service is bound to a particular IP address it will only accept connections directed to that address, whereas when bound to the wildcard address it will accept connections to any local address. Binding to the loopback address has the effect of prohibiting connections from other machines.

For most purposes the best way to construct the socket address is by calling `getaddrinfo`. This takes a string containing the IP address and a string containing the port number, and converts them into a `sockaddr_in` or a `sockaddr_in6` as appropriate. It is also able to resolve hostnames and service names:

```
const char* hostname=0; /* wildcard */
const char* portname="daytime";
struct addrinfo hints;
memset(&hints,0,sizeof(hints));
hints.ai_family=AF_UNSPEC;
hints.ai_socktype=SOCK_STREAM;
hints.ai_protocol=0;
hints.ai_flags=AI_PASSIVE|AI_ADDRCONFIG;
struct addrinfo* res=0;
int err=getaddrinfo(hostname,portname,&hints,&res);
if (err!=0) {
    die("failed to resolve local socket address (err=%d)",err);
}
```

The `hints` argument contains additional information to help guide the conversion. In this example:

- The address family has been left unspecified so that both IPv4 and IPv6 addresses can be returned. In principle you could receive results for other address families too: you can either treat this as a feature, or filter out any unwanted results after the call to `getaddrinfo`.
- The socket type has been constrained to `SOCK_STREAM`. This allows TCP but excludes UDP.

- The protocol has been left unspecified because it is only meaningful in the context of a specific address family. If the address family had been set to `AF_INET` or `AF_INET6` then this field could have been set to `IPPROTO_TCP` (but it is equally acceptable to leave it set to zero).
- The `AI_PASSIVE` flag has been set because the address is intended for binding to a server socket. It causes the IP address to default to the wildcard address as opposed to the loopback address.
- The `AI_ADDRCONFIG` flag has been set so that IPv6 results will only be returned if the server has an IPv6 address, and similarly for IPv4.

The `res` argument is used to return a linked list of `addrinfo` structures containing the address or addresses that were found. If the network service daemon has the ability to listen on multiple sockets then it should open one for each address in the list. Otherwise it is considered acceptable to use the first result and discard the remainder.

The memory occupied by the result list should be released by calling `freeaddrinfo` once it is no longer needed, however this cannot be done until after the socket has been created and bound.

Create the server socket

The socket that will be used to listen for connections should be created using the `socket` function. This takes three arguments:

1. the domain (`AF_INET` or `AF_INET6` in this case, corresponding to IPv4 or IPv6 respectively),
2. the socket type (`SOCK_STREAM` in this case, meaning that the socket should provide reliable transport of an unstructured byte stream), and
3. the protocol (`IPROTO_TCP` in this case, corresponding to TCP).

A value of 0 for the protocol requests the default for the given address family and socket type, which for `AF_INET` or `AF_INET6` and `SOCK_STREAM` would be `IPPROTO_TCP`. It is equally acceptable for the protocol to be deduced in this manner or specified explicitly.

Assuming you previously used `getaddrinfo` to construct the local address then the required values can be obtained from the `addrinfo` structure:

```
int server_fd=socket(res->ai_family,res->ai_socktype,res->ai_protocol);
if (server_fd==-1) {
    die("%s",strerror(errno));
}
```

Set the SO_REUSEADDR socket option

`SO_REUSEADDR` should be routinely set for TCP server sockets in order to allow the network service to be restarted when there are connections in the ESTABLISHED or TIME-WAIT state:

```
int reuseaddr=1;
if (setsockopt(server_fd,SOL_SOCKET,SO_REUSEADDR,&reuseaddr,sizeof(reuseaddr))==-1) {
    die("%s",strerror(errno));
}
```

See ‘Listen on a TCP port with connections in the TIME-WAIT state’ for a detailed discussion of this issue.

Bind the local address to the server socket

As noted previously, the server socket must be bound to a local address before it can listen for connections. This should be done using the `bind` function:

```
if (bind(server_fd,res->ai_addr,res->ai_addrlen)==-1) {
    die("%s",strerror(errno));
}
```

The first argument is the socket descriptor. The second and third arguments are the local address and its length.

If the local address was constructed using `getaddrinfo` then the memory occupied by the address list can now be released:

```
freeaddrinfo(res);
```

(If the address list has been searched or filtered then take care that it is the head of the list that is released, not the address that you have chosen to use.)

Listen for connections

The server socket can now be instructed to listen for connections. This should be done using the `listen` function:

```
if (listen(server_fd,SOMAXCONN)) {
    die("failed to listen for connections (errno=%d)",errno);
}
```

The first argument is the socket descriptor. The second argument is the backlog of outstanding connections that the operating system should queue while they are waiting to be accepted by the server process. It is only a hint: most implementations take some account of the value requested, but you should not make any assumptions. A value of `SOMAXCONN` indicates that the maximum permissible queue length should be selected.

The optimum value for the backlog depends on the nature of the load:

- If the value is too low then the server will be poor at handling short-term bursts of activity. Connections may be rejected even if the average load is well below what the server can handle.
- If the value is too high then the server will perform less well when it is genuinely overloaded. Under those circumstances, lengthening the queue merely increases latency without improving capacity.

A backlog of 5 is a popular choice due to its use in many tutorials. For services that receive connections at a very slow rate this is probably adequate, but it is too low for services that handle many short-lived connections (such as web servers). In that case the author's advice would be to make the value configurable, with a default of `SOMAXCONN`.

Accept connections as they arrive

Connections are accepted by the server process by repeatedly calling the `accept` function. Each time this is done a new socket descriptor is returned to act as an endpoint for the newly established connection. If no connections are available then the function blocks.

The process of handling a connection should preferably not interfere with the acceptance or handling of other connections. One way to ensure this is to spawn a new child process for each connection:

```
for (;;) {
    int session_fd=accept(server_fd,0,0);
    if (session_fd==-1) {
        if (errno==EINTR) continue;
        die("failed to accept connection (errno=%d)",errno);
    }
    pid_t pid=fork();
    if (pid==-1) {
        die("failed to create child process (errno=%d)",errno);
    } else if (pid==0) {
        close(server_fd);
        handle_session(session_fd);
        close(session_fd);
        _exit(0);
    } else {
        close(session_fd);
    }
}
```

The parent process should close the descriptor for each connected socket once the corresponding child process has been spawned. There are two reasons for doing this: to prevent the descriptors from accumulating, and to prevent the connection from being held open by the parent after it has been closed by the child. Similarly, the child process should close any file or socket descriptors inherited from the parent that it does not need access to. This will certainly include the descriptor for the server socket, but you should consider whether there are any others.

Functionality that is specific to the network service is represented here by the function `handle_session`. As a simple example, here is an implementation of the Daytime Protocol:

```
void handle_session(int session_fd) {
    time_t now=time(0);
    char buffer[80];
    size_t length=strftime(buffer,sizeof(buffer),"%a %b %d %T %Y\r\n",localtime(&now));
    if (length==0) {
        snprintf(buffer,sizeof(buffer),"Error: buffer overflow\r\n");
    }

    size_t index=0;
    while (index<length) {
        ssize_t count=write(session_fd,buffer+index,length-index);
        if (count<0) {
            if (errno==EINTR) continue;
            die("failed to write to socket (errno=%d)",errno);
        } else {
            index+=count;
        }
    }
}
```

Variations

Determining the remote address

It is often desirable and sometimes necessary to determine the remote address from which an inbound connection originated. A common reason for wanting to do this is to keep an log of all connections. Other possible motivations include access control, or establishing an outbound connection back to the client.

The address can be obtained at the time when the connection is accepted by supplying a buffer to place it in. Alternatively, it can be obtained at any time while the connection is open by calling `getpeername`.

The supplied buffer must be large enough and sufficiently well-aligned to accept any socket address that might be returned. If the address family has not been hard-coded then you can use the type `struct sockaddr_storage`, which is designed to hold addresses of any type:

```
struct sockaddr_storage sa;
socklen_t sa_len=sizeof(sa);
int session_fd=accept(server_fd,(struct sockaddr*)&sa,&sa_len);
```

Alternatively, if the local address was constructed using `getaddrinfo` then the required size in bytes can be found in the `ai_addrlen` member of the relevant `addrinfo` structure.

If there is a need to convert the address to human-readable form then this is best done using the `getnameinfo` function, especially if it is not known whether the address family is IPv4 or IPv6:

```
char buffer[INET6_ADDRSTRLEN];
int err=getnameinfo((struct sockaddr*)&sa,sa_len,buffer,sizeof(buffer),0,0,NI_NUMERICHOST);
if (err!=0) {
    snprintf(buffer,sizeof(buffer),"invalid address");
}
```

A useful refinement is to convert IPv4-mapped addresses into plain IPv4 addresses prior to calling `getnameinfo`:

```
if (sa.ss_family==AF_INET6) {
    struct sockaddr_in6* sa6=(struct sockaddr_in6*)&sa;
    if (IN6_IS_ADDR_V4MAPPED(&sa6->sin6_addr)) {
        struct sockaddr_in sa4;
        memset(&sa4,0,sizeof(sa4));
        sa4.sin_family=AF_INET;
        sa4.sin_port=sa6->sin6_port;
        memcpy(&sa4.sin_addr.s_addr,sa6->sin6_addr.s6_addr+12,4);
        memcpy(&sa,&sa4,sizeof(sa4));
        sa_len=sizeof(sa4);
    }
}
```

For example, if a IPv4 connection from 192.168.0.1 were received using an IPv6 socket then the code fragment above would cause the address to be presented as 192.168.0.1 instead of the less readable ::ffff:192.168.0.1.

Constructing the local socket address without using getaddrinfo

There are some circumstances where `getaddrinfo` is not he best way to construct the local socket address. For example, you may already have the port number and IP address in numeric form, or you may need to be compatible with older systems on which `getaddrinfo` is not available. A solution in these cases is to construct the socket address explicitly.

An IPv4 socket address is represented by a `struct sockaddr_in`. It should be zeroed before use, and any information within it should be stored in network byte order. For example, to create a socket address with a port number of 13 and the wildcard IP address:

```
struct sockaddr_in addr;
memset(&addr,0,sizeof(addr));
addr.sin_family=AF_INET;
addr.sin_port=htons(13);
addr.sin_addr.s_addr=htonl(INADDR_ANY);
```

Similarly for IPv6:

```
struct sockaddr_in6 addr;
memset(&addr,0,sizeof(addr));
addr.sin6_family=AF_INET6;
addr.sin6_flowinfo=0;
addr.sin6_port=htons(13);
addr.sin6_addr=in6addr_any;
```

See also

- Listen on a TCP port with connections in the TIME-WAIT state
- Establish a TCP connection in C
- Listen for and receive UDP datagrams in C

Further Reading

- Listen on a TCP port with connections in the TIME-WAIT state
- Convert an IP address to a human-readable string in C

Tags: c | posix | socket