Establish a TCP connection in C

Content

1 Objective
2 Scenario
3 Method
  3.1 Overview
  3.2 Construct the remote socket address
  3.3 Create the client socket
  3.4 Connect the socket to the remote address.
4 See also
5 Further Reading

Tested on

Debian (Lenny)
Ubuntu (Precise)

Objective

To establish an outbound TCP connection in C

Scenario

Suppose that you wish to write a client 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. The client is not required to send any data, and anything it does send is ignored.

Method

Overview

The method described here has three steps:

1. Construct the remote socket address.
2. Create the client socket.
3. Connect the socket to the remote address.

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 remote socket address

To establish an outbound TCP connection it is necessary to specify the remote IP address and port number to which the connection should be directed. 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.

(A local socket address may also be specified, however it is rarely necessary to do so. By default the local address is chosen automatically by the network stack.)

Most common network services have an assigned port number on which they are normally expected to listen. It makes sense for the client to use this as the default, however it is important that an alternative can be selected. The user of the client will not necessarily have any control over how the server is configured, so the onus is on the client software to provide access to whichever port the server has been instructed to use.

It is often useful for the remote IP address to default to the loopback address, particularly for services such as databases where there is a good chance of the client and server being run on the same machine. Alternatively, it is sometimes preferable to require that the destination be specified explicitly.

For most purposes the best way to construct the remote address is by calling `getaddrinfo`. This takes a string containing either a hostname or an IP address, and a second string containing either a service name or a port number. These are converted into a `sockaddr_in` or a `sockaddr_in6` as appropriate:

```
const char* hostname=0; /* localhost */
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_ADDRCONFIG;
struct addrinfo* res=0;
int err=getaddrinfo(hostname,portname,&hints,&res);
if (err!=0) {
    die("failed to resolve remote 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 not been set because the result is intended for use as a remote address. Its absence causes the IP address to default to the loopback address (as opposed to the wildcard 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 multiple records are returned then the recommended behaviour (from RFC 1123) is to

try each address in turn, stopping when a connection is successfully established. When doing this you may wish to limit the number of addresses tried and/or allow connection attempts to overlap, in order to prevent the cumulative timeout period from becoming excessive.

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 connected.

Create the client socket

The socket that will be used to establish the connection 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 remote address then the required values can be obtained from the `addrinfo` structure:

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

When iterating through a list of addresses returned by `getaddrinfo` it is potentially necessary to create a separate socket for each, because the addresses will not necessarily be members of the same address family or use the same protocol.

Connect the socket to the remote address.

A connection is established by calling the `connect` function:

```
if (connect(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 remote socket address and its length.

By default the `connect` function blocks until the initial TCP handshake has been completed and the socket is ready for use, or alternatively, until the connection attempt fails. Some types of connection failure are reported very quickly, whereas others can only be detected by means of a timeout. In the latter case `connect` may block for several minutes.

If the remote 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.)

The socket descriptor is now ready for use. Here is an example of how it might be utilised to implement a Daytime Protocol client:

```
char buffer[256];
for (;;) {
    ssize_t count=read(fd,buffer,sizeof(buffer));
    if (count<0) {
        if (errno!=EINTR) die("%s",strerror(errno));
    } else if (count==0) {
        break;
    } else {
        write(STDOUT_FILENO,buffer,count);
    }
}
close(fd);
```

See also

- Listen for and accept TCP connections in C
- Send a UDP datagram in C
- Send an arbitrary IPv4 datagram using a raw socket in C

Further Reading

- *Listen for and accept TCP connections in C*, microHOWTO

Tags: c | posix | socket