- The *Learning to use TCP/IP to communicate with processes on another machine* recipe to learn how to develop a connection-oriented program
- The *Learning to use UDP/IP to communicate with processes on another machine* recipe to learn how to develop a connectionless-oriented program

Learning to use TCP/IP to communicate with processes on another machine

This recipe will show you how to connect two programs by using a connection-oriented mechanism. This recipe will use TCP/IP, which is the *de facto* standard on the internet. So far, we've learned that TCP/IP is a reliable form of communication, and its connection is made in three phases. It is time now to write a program to learn how to make two programs communicate with each other. Although the language used will be C++, the communication part will be written using the Linux system calls, as it is not supported by the C++ standard library.

How to do it...

We'll develop two programs, a client and a server. The server will start and `listen` on a specific port that is ready to accept an incoming connection. The client will start and connect to the server identified by an IP and a port number:

1. With the Docker image running, open a shell and create a new file, `clientTCP.cpp`. Let's add some headers and constants that we'll need later:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <iostream>

constexpr unsigned int SERVER_PORT = 50544;
constexpr unsigned int MAX_BUFFER = 128;
```

2. Let's start writing the `main` method now. We start by initializing `socket` and getting the information that is related to the server:

```
int main(int argc, char *argv[])
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        std::cerr << "socket error" << std::endl;
        return 1;
    }
    struct hostent* server = gethostbyname(argv[1]);
    if (server == nullptr)
    {
        std::cerr << "gethostbyname, no such host" << std::endl;
        return 2;
    }
```

3. Next, we want to `connect` to the server, but we need the correct information, namely the `serv_addr`:

```
    struct sockaddr_in serv_addr;
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
          (char *)&serv_addr.sin_addr.s_addr,
          server->h_length);
    serv_addr.sin_port = htons(SERVER_PORT);
    if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof
        (serv_addr)) < 0)
    {
        std::cerr << "connect error" << std::endl;
        return 3;
    }
```

4. The server will reply with a connection `ack`, so we call the `read` method:

```
    std::string readBuffer (MAX_BUFFER, 0);
    if (read(sockfd, &readBuffer[0], MAX_BUFFER-1) < 0)
    {
        std::cerr << "read from socket failed" << std::endl;
        return 5;
    }
    std::cout << readBuffer << std::endl;
```

5. We can now send the data to the server by just calling the `write` system call:

```
        std::string writeBuffer (MAX_BUFFER, 0);
        std::cout << "What message for the server? : ";
        getline(std::cin, writeBuffer);
        if (write(sockfd, writeBuffer.c_str(), strlen(write
            Buffer.c_str())) < 0)
        {
            std::cerr << "write to socket" << std::endl;
            return 4;
        }
```

6. Finally, let's go through the cleaning part, where we have to close the socket:

```
        close(sockfd);
        return 0;
    }
```

7. Let's now develop the server program. In a second shell, we create the `serverTCP.cpp` file:

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

    constexpr unsigned int SERVER_PORT = 50544;
    constexpr unsigned int MAX_BUFFER = 128;
    constexpr unsigned int MSG_REPLY_LENGTH = 18;
```

8. On a second shell, first of all, we need a `socket` descriptor that will identify our connection:

```
    int main(int argc, char *argv[])
    {
         int sockfd =  socket(AF_INET, SOCK_STREAM, 0);
         if (sockfd < 0)
         {
              std::cerr << "open socket error" << std::endl;
```

```
     return 1;
}

int optval = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const
  void *)&optval , sizeof(int));
```

9. We have to bind the `socket` to a port and `serv_addr` on the local machine:

```
struct sockaddr_in serv_addr, cli_addr;
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(SERVER_PORT);
if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof
   (serv_addr)) < 0)
{
     std::cerr << "bind error" << std::endl;
     return 2;
}
```

10. Next, we have to wait for and accept any incoming connection:

```
listen(sockfd, 5);
socklen_t clilen = sizeof(cli_addr);
int newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
    &clilen);
if (newsockfd < 0)
{
     std::cerr << "accept error" << std::endl;
     return 3;
}
```

11. As soon as we get a connection, we log who connected to the standard output (using their IP and port) and send a confirmation *ACK*:

```
std::cout << "server: got connection from = "
          << inet_ntoa(cli_addr.sin_addr)
          << " and port = " << ntohs(cli_addr.sin_port)
             << std::endl;
write(incomingSock, "You are connected!", MSG_REPLY_LENGTH);
```

12. We made the connection (a three-way handshake, remember?), so now we can read any data coming from the client:

```
        std::string buffer (MAX_BUFFER, 0);
        if (read(incomingSock, &buffer[0], MAX_BUFFER-1) < 0)
        {
             std::cerr << "read from socket error" << std::endl;
             return 4;
        }
        std::cout << "Got the message:" << buffer << std::endl;
```

13. Finally, we close both the sockets:

```
        close(incomingSock);
        close(sockfd);
        return 0;
   }
```

We've written quite a lot of code, so it is time to explain how all of this works.

How it works...

Both the client and the server have a very common algorithm, which we have to describe in order for you to understand and generalize this concept. The client's algorithm is as follows:

```
socket() -> connect() -> send() -> receive()
```

Here, `connect()` and `receive()` are blocking calls (that is, the calling program will wait for their completion). The `connect` phrase specifically initiates the three-way handshake that we described in detail in the *Learning the basics of connection-oriented communication* recipe.

The server's algorithm is as follows:

```
socket() -> bind() -> listen() -> accept() -> receive() -> send()
```

Here, `accept` and `receive` are blocking the call. Let's now analyze in detail both the client's and server's code.

The client code analysis is as follows:

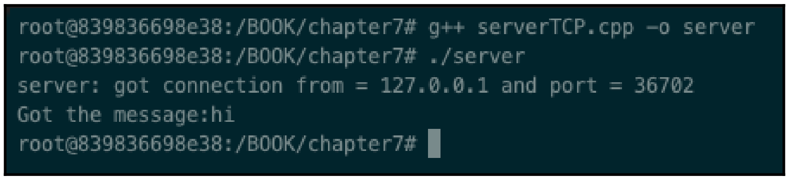
1. The first step just contains the necessary includes that are needed to correctly use the four APIs that we listed in the preceding client's algorithm section. Just note that the constants, in pure C++ style, are not defined using the `#define` macro, but by using `constexpr`. The difference is that the latter is managed by the compiler, whereas the former is managed by the preprocessor. As a rule of thumb, you should always try to rely on the compiler.
2. The `socket()` system call creates a socket descriptor that we named `sockfd`, which will be used to send and receive information to/from the server. The two parameters indicate that the socket will be a TCP (`SOCK_STREAM`)/IP (`PF_INET`) socket type. Once we have a valid socket descriptor, and before calling the `connect` method, we need to know the server's details; for this, we use the `gethostbyname()` method, which, given a string like `localhost`, will return a pointer to `struct hostent *` with information about the host.
3. We're now ready to call the `connect()` method, which will take care of the three-way-handshake process. By looking at its prototype (`man connect`), we can see that as well as the socket, it needs a `const struct sockaddr *address` struct, so we need to copy the respective information into it and pass it to the `connect()`; that's why we use the `utility` method `bcopy()` (`bzero()` is just a helper method to reset the `sockaddr` struct before using it).
4. We are now ready to send and receive data. Once the connection is established, the server will send an acknowledgment message (`You are connected!`). Have you noticed that we're using the `read()` method to receive information from the server through a socket? This is the beauty and simplicity of programming in a Linux environment. One method can support multiple interfaces—indeed, we're able to work with the same method to read files, receive data with sockets, and do many other things.
5. We can send a message to the server. The method used is, as you may have guessed, `write()`. We pass `socket` to it, which identifies the connection, the message we want the server to receive, and the length of the message so that Linux will know when to stop reading from the buffer.
6. As usual, we need to close, clean, and free any resource used. In this case, we have to close the socket by just using the `close()` method, passing the socket descriptor.

The server code analysis is as follows:

1. We use a similar code to the one we used for the client, but include some headers and three defined constants, which we will use and explain later.
2. We have to define a socket descriptor by calling the `socket()` API. Note that there is no difference between the client and the server. We just need a socket that is able to manage a TCP/IP type of a connection.
3. We have to bind the socket descriptor created in the previous step to the network interface and port it on the local machine. We do this with the `bind()` method, which assigns an address (`const struct sockaddr *address` passed as the second parameter) to the socket descriptor passed as the first parameter. The call to the `setsockopt()` method is just to avoid the bind error, `Address already in use`.
4. We start listening for any incoming connection by calling the `listen()` API. The `listen()` system call is pretty simple: it gets the `socket` descriptor on which we are listening and the maximum number of connections to keep in the queue of pending connections, which in our case we set to `5`. Then we call `accept()` on the socket descriptor. The `accept` method is a blocking call: it means that it'll block until a new incoming connection is available, and then it'll return an integer representing the socket descriptor. The `cli_addr` structure is filled in with the connection's information, which we use to log who connected (`IP` and `port`).
5. This step is just a logical continuation of step 10. Once the server accepts a connection, we log on the standard output who connected (in terms of their `IP` and `port`). We do this by querying the information that was filled in the `cli_addr` struct by the `accept` method.
6. In this step, we receive information from the connected client through the `read()` system call. We pass in the input, the socket descriptor of the incoming connection, the `buffer` where the data will be saved, and the maximum length of the data that we want to read (`MAX_BUFFER-1`).
7. We then clean up and free any eventual resource that is used and/or allocated. In this case, we have to close the two sockets' descriptors that were used (`sockfd` for the server and `incomingSock` for the incoming connection).

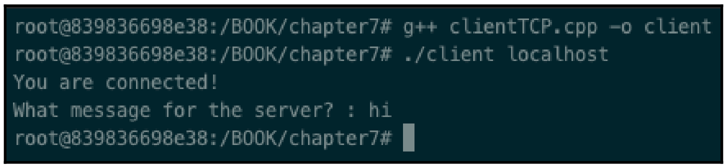
By building and running both the server and the client (in this order), we get the following output:

- The server build and output are as follows:



```
root@839836698e38:/BOOK/chapter7# g++ serverTCP.cpp -o server
root@839836698e38:/BOOK/chapter7# ./server
server: got connection from = 127.0.0.1 and port = 36702
Got the message:hi
root@839836698e38:/BOOK/chapter7# 
```

- The client build and output are as follows:



```
root@839836698e38:/BOOK/chapter7# g++ clientTCP.cpp -o client
root@839836698e38:/BOOK/chapter7# ./client localhost
You are connected!
What message for the server? : hi
root@839836698e38:/BOOK/chapter7# 
```

This proves what we learned in this recipe.

There's more...

How can we improve the server application to manage multiple concurrent incoming connections? The server's algorithm that we implemented is sequential; after `listen()`, we just wait on `accept()` until the end, where we close the connections. You should go through the following steps as an exercise:

1. Run an infinite loop over `accept()` so that a server is always up and ready to serve clients.
2. Spin off a new thread for each accepted connection. You can do this by using `std::thread` or `std::async`.

Another important practice is to pay attention to the data that the client and server exchange with each other. Usually, they agree to use a protocol that they both know. It might be a web server, which in that case will involve the exchange of HTML, files, resources, and so on between the client and the server. If it is a supervision and control system, it might be a protocol defined by a specific standard.