Boost.Asio C++ Network Programming Cookbook

Boost.Asio is a powerful cross-platform C++ library designed by experts in the field to reduce complexity and simplify network programming, while ensuring maximum efficiency.

Starting with recipes demonstrating the execution of basic Boost.Asio operations, the book goes on to provide ready-to-use implementations of client and server applications, from simple synchronous ones to powerful multithreaded scalable solutions. Finally, the reader is presented with advanced topics such as implementing asynchronous TCP server, implementing an HTTP client and server, and adding SSL/TLS support. All samples presented in the book are ready to be used in real projects out of the box. Apart from excellent practical examples, the book also includes extended supporting theoretical material on distributed application design and construction.

What this book will do for you...

- Boost your working knowledge of one of the most reputable C++ networking libraries—Boost.Asio
- Familiarize yourself with the basics of TCP and UDP protocols
- Create scalable and highly-efficient client and server applications
- Understand the theory behind the development of distributed applications
- Increase the security of your distributed applications by adding SSL/TLS support
- Implement an HTTP client and server easily
- Use iostreams, scatter-gather buffers, and timers

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Over 25 hands-on recipes to create robust and highly-efficient cross-platform distributed applications with the Boost.Asio library

Dmytro Radchuk
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'The Basics'
- A synopsis of the book’s content
- More information on boost.asio C++ Network Programming Cookbook
About the Author

Dmytro Radchuk is a software engineer from Kyiv, Ukraine. His passion for science started at a young age and led him to receive a degree in computer science from Kyiv Polytechnic University. As of today, he has deep technical knowledge and more than 8 years of experience in the field of software development for several industries and businesses. He strongly believes that the whole is greater than the sum of its parts and this is one of the reasons why the development of distributed applications has become his main specialization.

Dmytro has always supported the idea of sharing knowledge and this has resulted in Boost. Asio C++ Network Programming Cookbook, which has become his first officially published book. He believes that science will help us deal with the monotony of everyday life. When he is not exploring another facet of computer engineering, he is probably learning a new aspect of psychology, history, or the arts, which are also of great interest to him.
Preface

In today's information-centric globalized world, telecommunications have become an essential part of our lives. They penetrate and play crucial roles in almost every aspect of our day-to-day activities, from personal to professional. Sometimes, a failure to communicate information correctly and on time may lead to significant loss of material assets or even casualties.

Therefore, it is very important to provide the highest level of reliability when it comes to developing telecommunication software. However, it can be a really challenging task due to the inherent complexity of the domain and accidental complexity of the low-level tools provided by modern operating systems.

The Boost.Asio library is aimed at reducing accidental complexity by introducing type systems and exploiting object-oriented methods, and decreasing the development time by providing high degrees of reusability. In addition to this, because the library is cross-platform, the applications implemented with it can be built on multiple platforms, which enhances software qualities even more, while decreasing its costs.

This book contains more than 30 recipes—step-by-step solutions to various tasks that often (and not so often) arise in the area of network programming. All recipes take advantage of facilities provided by the Boost.Asio library, demonstrating best practices of applying the library to execute typical tasks and solve different problems.

What this book covers

Chapter 1, The Basics, introduces you to basic classes provided by the Boost.Asio library and demonstrates how to execute basic operations, such as resolving a DNS name, connecting a socket, accepting a connection, and so on.

Chapter 2, I/O Operations, demonstrates how to perform individual network I/O operations, both synchronous and asynchronous.
Chapter 3, Implementing Client Applications, contains recipes that demonstrate how to implement different types of client applications.

Chapter 4, Implementing Server Applications, contains recipes that demonstrate how to implement different types of server applications.

Chapter 5, HTTP and SSL/TLS, covers more advanced topics on the HTTP and SSL/TLS protocol implementation.

Chapter 6, Other Topics, includes recipes that discuss less popular but still quite important topics, such as timers, socket options, composite buffers, and others.
1
The Basics

In this chapter, we will cover:

- Creating an endpoint
- Creating an active socket
- Creating a passive socket
- Resolving a DNS name
- Binding a socket to an endpoint
- Connecting a socket
- Accepting connections

Introduction

Computer networks and communication protocols significantly increase capabilities of modern software, allowing different applications or separate parts of the same application to communicate with each other to achieve a common goal. Some applications have communication as their main function, for example, instant messengers, e-mail servers and clients, file download software, and so on. Others have the network communication layer as a fundamental component, on top of which the main functionality is built. Some of the examples of such applications are web browsers, network file systems, distributed database management systems, media streaming software, online games, offline games with multiplayer over the network option support, and many others. Besides, nowadays almost any application in addition to its main functionality provides supplementary functions, involving network communication. The most prominent examples of such functions are online registration and automatic software update. In the latter case, the update package is downloaded from the application developer's remote server and installed on the user's computer or mobile device.
The application that consists of two or more parts, each of which runs on a separate computing device, and communicates with other parts over a computer network is called a **distributed application**. For example, a web server and a web browser together can be considered as one complex distributed application. The browser running on a user's computer communicates with the web server running on a different remote computer in order to achieve a common goal—to transmit and display a web page requested by the user.

Distributed applications provide significant benefits as compared to traditional applications running on a single computer. The most valuable of them are the following:

- Ability to transmit data between two or more remote computing devices. This is absolutely obvious and the most valuable benefit of distributed software.
- Ability to connect computers in a network and install special software on them, creating powerful computing systems that can perform tasks that can't otherwise be performed on a single computer in an adequate amount of time.
- Ability to effectively store and share data in a network. In a computer network, a single device can be used as data storage to store big amounts of data and other devices can easily request some portions of that data when necessary without the need to keep the copy of all data on each device. As an example, consider large datacenters hosting hundreds of millions of websites. The end user can request the web page they need anytime by sending the request to the server over the network (usually, the Internet). There is no need to keep the copy of the website on the user's device. There is a single storage of the data (a website) and millions of users can request the data from that storage if and when this information is needed.

For two applications running on different computing devices to communicate with each other, they need to agree on a communication protocol. Of course, the developer of the distributed application is free to implement his or her own protocol. However, this would be rarely the case at least for two reasons. First, developing such a protocol is an enormously complex and time-consuming task. Secondly, such protocols are already defined, standardized, and even implemented in all popular operating systems including Windows, Mac OS X, and majority of the distributions of Linux.

These protocols are defined by the TCP/IP standard. Don't be fooled by the standard's name; it defines not only TCP and IP but many more other protocols, comprising a TCP/IP protocol stack with one or more protocols on each level of the stack. Distributed software developers usually deal with transport level protocols such as TCP or UDP. Lower layer protocols are usually hidden from the developer and are handled by the operating system and network devices.
In this book, we only touch upon TCP and UDP protocols that satisfy the needs of most developers of distributed software. If the reader is not familiar with the TCP/IP protocol stack, the OSI model, or TCP and UDP protocols, it's highly advised to read some theory on these topics. Though this book provides some brief information about them, it is mostly focused on practical aspects of using TCP and UDP protocols in distributed software development.

The TCP protocol is a transport layer protocol with the following characteristics:

- It's reliable, which means that this protocol guarantees delivery of the messages in proper order or a notification that the message has not been delivered. The protocol includes error handling mechanisms, which frees the developer from the need to implement them in the application.
- It assumes logical connection establishment. Before one application can communicate with another over the TCP protocol, it must establish a logical connection by exchanging service messages according to the standard.
- It assumes the point-to-point communication model. That is, only two applications can communicate over a single connection. No multicast messaging is supported.
- It is stream-oriented. This means that the data being sent by one application to another is interpreted by the protocol as a stream of bytes. In practice, it means that if a sender application sends a particular block of data, there is no guarantee that it will be delivered to the receiver application as the same block of data in a single turn, that is, the sent message may be broken into as many parts as the protocol wants and each of them will be delivered separately, though in correct order.

The UDP protocol is a transport layer protocol having different (in some sense opposite) characteristics from those of the TCP protocol. The following are its characteristics:

- It's unreliable, which means that if a sender sends a message over a UDP protocol, there is no guarantee that the message will be delivered. The protocol won't try to detect or fix any errors. The developer is responsible for all error handling.
- It's connectionless, meaning that no connection establishment is needed before the applications can communicate.
- It supports both one-to-one and one-to-many communication models. Multicast messages are supported by the protocol.
- It's datagram oriented. This means that the protocol interprets data as messages of a particular size and will try to deliver them as a whole. The message (datagram) either will be delivered as a whole, or if the protocol fails to do that won't be delivered at all.

Because the UDP protocol is unreliable, it is usually used in reliable local networks. To use it for communication over the Internet (which is an unreliable network), the developer must implement error handling mechanisms in its application.
When there is a need to communicate over the Internet, the TCP protocol is most often the best choice due to its reliability.

As it has already been mentioned, both TCP and UDP protocols and the underlying protocols required by them are implemented by most popular operating systems. A developer of a distributed application is provided an API through which it can use protocols implementation. The TCP/IP standard does not standardize the protocol API implementation; therefore, several API implementations exist. However, the one based on Berkeley Sockets API is the most widely used.

Berkeley Sockets API is the name of one of the many possible implementations of TCP and UDP protocols' API. This API was developed at the Berkeley University of California, USA (hence the name) in the early 1980s. It is built around a concept of an abstract object called a socket. Such a name was given to this object in order to draw the analogy with a usual electrical socket. However, this idea seems to have somewhat failed due to the fact that Berkeley Sockets turned out to be a significantly more complex concept.

Now Windows, Mac OS X, and Linux operating systems all have this API implemented (though with some minor variations) and software developers can use it to consume TCP and UDP protocols' functionality when developing distributed applications.

Though very popular and widely used, Sockets API has several flaws. First, because it was designed as a very generic API that should support many different protocols, it is quite complex and somewhat difficult to use. The second flaw is that this is a C-style functional API with a poor type system, which makes it error prone and even more difficult to use. For example, Sockets API doesn't provide a separate type representing a socket. Instead, the built-in type int is used, which means that by mistake any value of the int type can be passed as an argument to the function expecting a socket, and the compiler won't detect the mistake. This may lead to run-time crashes, the root cause of which is hard to find.
Network programming is inherently complex and doing it with a low-level C-style socket API makes it even more complex and error prone. Boost.Asio is an O-O C++ library that is, just like raw Sockets API, built around the concept of a socket. Roughly speaking, Boost.Asio wraps raw Sockets API and provides the developer with O-O interface to it. It is intended to simplify network programming in several ways as follows:

- It hides the raw C-style API and providing a user with an object-oriented API
- It provides a rich-type system, which makes code more readable and allows it to catch many errors at compilation time
- As Boost.Asio is a cross-platform library, it simplifies development of cross-platform distributed applications
- It provides auxiliary functionality such as scatter-gather I/O operations, stream-based I/O, exception-based error handling, and others
- The library is designed so that it can be relatively easily extended to add new custom functionality

This chapter introduces essential Boost.Asio classes and demonstrates how to perform basic operations with them.

Creating an endpoint

A typical client application, before it can communicate with a server application to consume its services, must obtain the IP address of the host on which the server application is running and the protocol port number associated with it. A pair of values consisting of an IP address and a protocol port number that uniquely identifies a particular application running on a particular host in a computer network is called an endpoint.

The client application will usually obtain the IP address and the port number identifying the server application either from the user directly through the application UI or as command-line arguments or will read it from the application’s configuration file.

The IP address can be represented as a string containing an address in dot-decimal notation if it is an IPv4 address (for example, 192.168.10.112) or in hexadecimal notation if it is an IPv6 address (for example, FE36::0404:C3FA:EF1E:3829). Besides, the server IP address can be provided to the client application in an indirect form, as a string containing a DNS name (for example, localhost or www.google.com). Another way to represent an IP address is an integer value. The IPv4 address is represented as a 32-bit integer and IPv6 as a 64-bit integer. However, due to poor readability and memorability this representation is used extremely rarely.
If the client application is provided with a DNS name before it can communicate with the server application, it must resolve the DNS name to obtain the actual IP address of the host running the server application. Sometimes, the DNS name may map to multiple IP addresses, in which case the client may want to try addresses one by one until it finds the one that works. We'll consider a recipe describing how to resolve DNS names with Boost.Asio later in this chapter.

The server application needs to deal with endpoints too. It uses the endpoint to specify to the operating system on which the IP address and protocol port it wants to listen for incoming messages from the clients. If the host running the server application has only one network interface and a single IP address assigned to it, the server application has only one option as to on which address to listen. However, sometimes the host might have more than one network interface and correspondingly more than one IP address. In this situation, the server application encounters a difficult problem of selecting an appropriate IP address on which to listen for incoming messages. The problem is that the application knows nothing about details such as underlying IP protocol settings, packet routing rules, DNS names which are mapped to the corresponding IP addresses, and so on. Therefore, it is quite a complex task (and sometimes even not solvable) for the server application to foresee through which IP address the messages sent by clients will be delivered to the host.

If the server application chooses only one IP address to listen for incoming messages, it may miss messages routed to other IP addresses of the host. Therefore, the server application usually wants to listen on all IP addresses available on the host. This guarantees that the server application will receive all messages arriving at any IP address and the particular protocol port.

To sum up, the endpoints serve two goals:

- The client application uses an endpoint to designate a particular server application it wants to communicate with.
- The server application uses an endpoint to specify a local IP address and a port number on which it wants to receive incoming messages from clients. If there is more than one IP address on the host, the server application will want to create a special endpoint representing all IP addresses at once.

This recipe explains how to create endpoints in Boost.Asio both in client and server applications.

**Getting ready**

Before creating the endpoint, the client application must obtain the raw IP address and the protocol port number designating the server it will communicate with. The server application on the other hand, as it usually listens for incoming messages on all IP addresses, only needs to obtain a port number on which to listen.
Here, we don't consider how the application obtains a raw IP address or a port number. In the following recipes, we assume that the IP address and the port number have already been obtained by the application and are available at the beginning of the corresponding algorithm.

**How to do it...**

The following algorithms and corresponding code samples demonstrate two common scenarios of creating an endpoint. The first one demonstrates how the client application can create an endpoint to specify the server it wants to communicate with. The second one demonstrates how the server application creates an endpoint to specify on which IP addresses and port it wants to listen for incoming messages from clients.

**Creating an endpoint in the client to designate the server**

The following algorithm describes steps required to perform in the client application to create an endpoint designating a server application the client wants to communicate with. Initially, the IP address is represented as a string in the dot-decimal notation if this is an IPv4 address or in hexadecimal notation if this is an IPv6 address:

1. Obtain the server application's IP address and port number. The IP address should be specified as a string in the dot-decimal (IPv4) or hexadecimal (IPv6) notation.
2. Represent the raw IP address as an object of the `asio::ip::address` class.
3. Instantiate the object of the `asio::ip::tcp::endpoint` class from the address object created in step 2 and a port number.
4. The endpoint is ready to be used to designate the server application in Boost.Asio communication related methods.

The following code sample demonstrates possible implementation of the algorithm:

```cpp
#include <boost/asio.hpp>
#include <iostream>
using namespace boost;

int main()
{
    // Step 1. Assume that the client application has already obtained the IP-address and the protocol port number.
    std::string raw_ip_address = "127.0.0.1";
    unsigned short port_num = 3333;

    // Used to store information about error that happens while parsing the raw IP-address.
    boost::system::error_code ec;
```
Creating the server endpoint

The following algorithm describes steps required to perform in a server application to create an endpoint specifying all IP addresses available on the host and a port number on which the server application wants to listen for incoming messages from the clients:

1. Obtain the protocol port number on which the server will listen for incoming requests.
2. Create a special instance of the `asio::ip::address` object representing all IP addresses available on the host running the server.
3. Instantiate an object of the `asio::ip::tcp::endpoint` class from the address object created in step 2 and a port number.
4. The endpoint is ready to be used to specify to the operating system that the server wants to listen for incoming messages on all IP addresses and a particular protocol port number.

The following code sample demonstrates possible implementation of the algorithm. Note that it is assumed that the server application is going to communicate over the IPv6 protocol:

```cpp
#include <boost/asio.hpp>
#include <iostream>
```
Chapter 1

```cpp
using namespace boost;

int main()
{
    // Step 1. Here we assume that the server application has
    // already obtained the protocol port number.
    unsigned short port_num = 3333;

    // Step 2. Create special object of asio::ip::address class
    // that specifies all IP-addresses available on the host. Note
    // that here we assume that server works over IPv6 protocol.
    asio::ip::address ip_address = asio::ip::address_v6::any();

    // Step 3.
    asio::ip::tcp::endpoint ep(ip_address, port_num);

    // Step 4. The endpoint is created and can be used to
    // specify the IP addresses and a port number on which
    // the server application wants to listen for incoming
    // connections.

    return 0;
}
```

**How it works...**

Let's consider the first code sample. The algorithm it implements is applicable in an application playing a role of a client that is an application that actively initiates the communication session with a server. The client application needs to be provided an IP address and a protocol port number of the server. Here we assume that those values have already been obtained and are available at the beginning of the algorithm, which makes step 1 details a given.

Having obtained the raw IP address, the client application must represent it in terms of the Boost.Asio type system. Boost.Asio provides three classes used to represent an IP address:

- `asio::ip::address_v4`: This represents an IPv4 address
- `asio::ip::address_v6`: This represents an IPv6 address
- `asio::ip::address`: This IP-protocol-version-agnostic class can represent both IPv4 and IPv6 addresses
In our sample, we use the `asio::ip::address` class, which makes the client application IP-protocol-version-agnostic. This means that it can transparently work with both IPv4 and IPv6 servers.

In step 2, we use the `asio::ip::address` class's static method, `from_string()`. This method accepts a raw IP address represented as a string, parses and validates the string, instantiates an object of the `asio::ip::address` class, and returns it to the caller. This method has four overloads. In our sample we use this one:

```cpp
class asio::ip::address

static asio::ip::address from_string(
    const std::string & str,
    boost::system::error_code & ec);
```

This method is very useful as it checks whether the string passed to it as an argument contains a valid IPv4 or IPv6 address and if it does, instantiates a corresponding object. If the address is invalid, the method will designate an error through the second argument. It means that this function can be used to validate the raw user input.

In step 3, we instantiate an object of the `boost::asio::ip::tcp::endpoint` class, passing the IP address and a protocol port number to its constructor. Now, the `ep` object can be used to designate a server application in the Boost.Asio communication related functions.

The second sample has a similar idea, although it somewhat differs from the first one. The server application is usually provided only with the protocol port number on which it should listen for incoming messages. The IP address is not provided because the server application usually wants to listen for the incoming messages on all IP addresses available on the host, not only on a specific one.

To represent the concept of all IP addresses available on the host, the classes `asio::ip::address_v4` and `asio::ip::address_v6` provide a static method `any()`, which instantiates a special object of corresponding class representing the concept. In step 2, we use the `any()` method of the `asio::ip::address_v6` class to instantiate such a special object.

Note that the IP-protocol-version-agnostic class `asio::ip::address` does not provide the `any()` method. The server application must explicitly specify whether it wants to receive requests either on IPv4 or on IPv6 addresses by using the object returned by the `any()` method of either the `asio::ip::address_v4` or `asio::ip::address_v6` class correspondingly. In step 2 of our second sample, we assume that our server communicates over IPv6 protocol and therefore called the `any()` method of the `asio::ip::address_v6` class.

In step 3, we create an endpoint object which represents all IP addresses available on the host and a particular protocol port number.
There's more...

In both our previous samples we used the endpoint class declared in the scope of the asio::ip::tcp class. If we look at the declaration of the asio::ip::tcp class, we'll see something like this:

```cpp
class tcp
{
    public:
        /// The type of a TCP endpoint.
        typedef basic_endpoint<tcp> endpoint;

        //...
}
```

It means that this endpoint class is a specialization of the basic_endpoint<> template class that is intended for use in clients and servers communicating over the TCP protocol.

However, creating endpoints that can be used in clients and servers that communicate over the UDP protocol is just as easy. To represent such an endpoint, we need to use the endpoint class declared in the scope of the asio::ip::udp class. The following code snippet demonstrates how this endpoint class is declared:

```cpp
class udp
{
    public:
        /// The type of a UDP endpoint.
        typedef basic_endpoint<udp> endpoint;

        //...
}
```

For example, if we want to create an endpoint in our client application to designate a server with which we want to communicate over the UDP protocol, we would only slightly change the implementation of step 3 in our sample. This is how that step would look like with changes highlighted:

```cpp
// Step 3.
asio::ip::udp::endpoint ep(ip_address, port_num);
```

All other code would not need to be changed as it is transport protocol independent.

The same trivial change in the implementation of step 3 in our second sample is required to switch from a server communicating over TCP to one communicating over UDP.
See also

- The Binding a socket to an endpoint recipe explains how the endpoint object is used in a server application
- The Connecting a socket recipe explains how the endpoint object is used in a client application

Creating an active socket

The TCP/IP standard tells us nothing about sockets. Moreover, it tells us almost nothing about how to implement the TCP or UDP protocol software API through which this software functionality can be consumed by the application.

If we look at section 3.8, Interface, of the RFC document #793 which describes the TCP protocol, we’ll find out that it contains only functional requirements of a minimal set of functions that the TCP protocol software API must provide. A developer of the protocol software is given full control over all other aspects of the API, such as the structure of the API, names of the functions comprising the API, the object model, the abstractions involved, additional auxiliary functions, and so on. Every developer of the TCP protocol software is free to choose the way to implement the interface to his or her protocol implementation.

The same story applies with the UDP protocol: only a small set of functional requirements of mandatory operations are described in the RFC document #768 devoted to it. The control of all other aspects of the UDP protocol software API is reserved for the developer of this API.

As it has already been mentioned in the introduction to this chapter, Berkeley Sockets API is the most popular TCP and UDP protocols’ API. It is designed around the concept of a socket—an abstract object representing a communication session context. Before we can perform any network I/O operations, we must first allocate a socket object and then associate each I/O operation with it.

Boost.Asio borrows many concepts from Berkeley Sockets API and is so much similar to it that we can call it “an object oriented Berkeley Sockets API”. The Boost.Asio library includes a class representing a socket concept, which provides interface methods similar to those found in Berkeley Sockets API.

Basically, there are two types of sockets. A socket intended to be used to send and receive data to and from a remote application or to initiate a connection establishment process with it is called an active socket, whereas a passive socket is the one used to passively wait for incoming connection requests from remote applications. Passive sockets don't take part in user data transmission. We'll talk about passive sockets later in this chapter.

This recipe explains how to create and open an active socket.
How to do it...

The following algorithm describes the steps required to perform in a client application to create and open an active socket:

1. Create an instance of the `asio::io_service` class or use the one that has been created earlier.
2. Create an object of the class that represents the transport layer protocol (TCP or UDP) and the version of the underlying IP protocol (IPv4 or IPv6) over which the socket is intended to communicate.
3. Create an object representing a socket corresponding to the required protocol type. Pass the object of `asio::io_service` class to the socket's constructor.
4. Call the socket's `open()` method, passing the object representing the protocol created in step 2 as an argument.

The following code sample demonstrates possible implementation of the algorithm. It is assumed that the socket is intended to be used to communicate over the TCP protocol and IPv4 as the underlying protocol:

```cpp
#include <boost/asio.hpp>
#include <iostream>

using namespace boost;

int main()
{
    // Step 1. An instance of 'io_service' class is required by
    // socket constructor.
    asio::io_service ios;

    // Step 2. Creating an object of 'tcp' class representing
    // a TCP protocol with IPv4 as underlying protocol.
    asio::ip::tcp protocol = asio::ip::tcp::v4();

    // Step 3. Instantiating an active TCP socket object.
    asio::ip::tcp::socket sock(ios);

    // Used to store information about error that happens
    // while opening the socket.
    boost::system::error_code ec;
```
The Basics

// Step 4. Opening the socket.
sock.open(protocol, ec);

if (ec.value() != 0) {
    // Failed to open the socket.
    std::cout
        << "Failed to open the socket! Error code = "
        << ec.value() << ". Message: " << ec.message();
    return ec.value();
}
return 0;
}

How it works...

In step 1, we instantiate an object of the asio::io_service class. This class is a central component in the Boost.Asio I/O infrastructure. It provides access to the network I/O services of the underlying operating system. Boost.Asio sockets get access to those services through the object of this class. Therefore, all socket class constructors require an object of asio::io_service as an argument. We'll consider the asio::io_service class in more detail in the following chapters.

In the next step, we create an instance of the asio::ip::tcp class. This class represents a TCP protocol. It provides no functionality, but rather acts like a data structure that contains a set of values that describe the protocol.

The asio::ip::tcp class doesn't have a public constructor. Instead, it provides two static methods, asio::ip::tcp::v4() and asio::ip::tcp::v6(), that return an object of the asio::ip::tcp class representing the TCP protocol with the underlying IPv4 or IPv6 protocol correspondingly.

Besides, the asio::ip::tcp class contains declarations of some basic types intended to be used with the TCP protocol. Among them are asio::tcp::endpoint, asio::tcp::socket, asio::tcp::acceptor, and others. Let's have a look at those declarations found in the boost/asio/ip/tcp.hpp file:

namespace boost {
    namespace asio {
        namespace ip {

        // ...
In step 3, we create an instance of the `asio::ip::tcp::socket` class, passing the object of the `asio::io_service` class to its constructor as an argument. Note that this constructor does not allocate the underlying operating system's socket object. The real operating system's socket is allocated in step 4 when we call the `open()` method and pass an object specifying protocol to it as an argument.

In Boost.Asio, **opening** a socket means associating it with full set of parameters describing a specific protocol over which the socket is intended to be communicating. When the Boost.Asio socket object is provided with these parameters, it has enough information to allocate a real socket object of the underlying operating system.

The `asio::ip::tcp::socket` class provides another constructor that accepts a protocol object as an argument. This constructor constructs a socket object and opens it. Note that this constructor throws an exception of the type `boost::system::system_error` if it fails. Here is a sample demonstrating how we could combine steps 3 and 4 from the previous sample:

```cpp
try {
    // Step 3 + 4 in single call. May throw.
    asio::ip::tcp::socket sock(ios, protocol);
} catch (boost::system::system_error & e) {
    std::cout << "Error occurred! Error code = " << e.code() << ". Message: " << e.what();
}
```
There's more...

The previous sample demonstrates how to create an active socket intended to communicate over the TCP protocol. The process of creating a socket intended for communication over the UDP protocol is almost identical.

The following sample demonstrates how to create an active UDP socket. It is assumed that the socket is going to be used to communicate over the UDP protocol with IPv6 as the underlying protocol. No explanation is provided with the sample because it is very similar to the previous one and therefore should not be difficult to understand:

```cpp
#include <boost/asio.hpp>
#include <iostream>

using namespace boost;

int main()
{
    // Step 1. An instance of 'io_service' class is required by
    // socket constructor.
    asio::io_service ios;

    // Step 2. Creating an object of 'udp' class representing
    // a UDP protocol with IPv6 as underlying protocol.
    asio::ip::udp protocol = asio::ip::udp::v6();

    // Step 3. Instantiating an active UDP socket object.
    asio::ip::udp::socket sock(ios);

    // Used to store information about error that happens
    // while opening the socket.
    boost::system::error_code ec;

    // Step 4. Opening the socket.
    sock.open(protocol, ec);

    if (ec.value() != 0) {
        // Failed to open the socket.
        std::cout
            << "Failed to open the socket! Error code = "
            << ec.value() << ". Message: " << ec.message();
        return ec.value();
    }

    return 0;
}
```
Creating a passive socket

A passive socket or acceptor socket is a type of socket that is used to wait for connection establishment requests from remote applications that communicate over the TCP protocol. This definition has two important implications:

- Passive sockets are used only in server applications or hybrid applications that may play both roles of the client and server.
- Passive sockets are defined only for the TCP protocol. As the UDP protocol doesn’t imply connection establishment, there is no need for a passive socket when communication is performed over UDP.

This recipe explains how to create and open a passive socket in Boost.Asio.

How to do it...

In Boost.Asio a passive socket is represented by the `asio::ip::tcp::acceptor` class. The name of the class suggests the key function of the objects of the class—to listen for and accept or handle incoming connection requests.

The following algorithm describes the steps required to perform to create an acceptor socket:

1. Create an instance of the `asio::io_service` class or use the one that has been created earlier.
2. Create an object of the `asio::ip::tcp` class that represents the TCP protocol and the required version of the underlying IP protocol (IPv4 or IPv6).
3. Create an object of the `asio::ip::tcp::acceptor` class representing an acceptor socket, passing the object of the `asio::io_service` class to its constructor.
4. Call the acceptor socket's `open()` method, passing the object representing the protocol created in step 2 as an argument.
The following code sample demonstrates the possible implementation of the algorithm. It is assumed that the acceptor socket is intended to be used over the TCP protocol and IPv6 as the underlying protocol:

```cpp
#include <boost/asio.hpp>
#include <iostream>

using namespace boost;

int main()
{
    // Step 1. An instance of 'io_service' class is required by
    // socket constructor.
    asio::io_service ios;

    // Step 2. Creating an object of 'tcp' class representing
    // a TCP protocol with IPv6 as underlying protocol.
    asio::ip::tcp protocol = asio::ip::tcp::v6();

    // Step 3. Instantiating an acceptor socket object.
    asio::ip::tcp::acceptor acceptor(ios);

    // Used to store information about error that happens
    // while opening the acceptor socket.
    boost::system::error_code ec;

    // Step 4. Opening the acceptor socket.
    acceptor.open(protocol, ec);

    if (ec.value() != 0) {
        // Failed to open the socket.
        std::cout
            << "Failed to open the acceptor socket!"
            << "Error code = " << ec.value() << ". Message: " << ec.message();
        return ec.value();
    }

    return 0;
}
```
How it works...

Because an acceptor socket is very similar to an active socket, the procedure of creating them is almost identical. Therefore, here we only shortly go through the sample code. For more details about each step and each object involved in the procedure, please refer to the Creating an active socket recipe.

In step 1, we create an instance of the asio::io_service class. This class is needed by all Boost.Asio components that need access to the services of the underlying operating system.

In step 2, we create an object representing a TCP protocol with IPv6 as its underlying protocol.

Then in step 3, we create an instance of the asio::ip::tcp::acceptor class, passing an object of the asio::io_service class as an argument to its constructor. Just as in the case of an active socket, this constructor instantiates an object of Boost.Asio the asio::ip::tcp::acceptor class, but does not allocate the actual socket object of the underlying operating system.

The operating system socket object is allocated in step 4, where we open the acceptor socket object, calling its open() method and passing the protocol object to it as an argument. If the call succeeds, the acceptor socket object is opened and can be used to start listening for incoming connection requests. Otherwise, the ec object of the boost::system::error_code class will contain error information.

See also

- The Creating an active socket recipe provides more details about the asio::io_service and asio::ip::tcp classes

Resolving a DNS name

Raw IP addresses are very inconvenient for humans to perceive and remember, especially if they are IPv6 addresses. Take a look at 192.168.10.123 (IPv4) or 8fee:9930:4545:a:105:f8ff:fe21:67cf (IPv6). Remembering those sequences of numbers and letters could be a challenge for anyone.

To enable labeling the devices in a network with human-friendly names, the Domain Name System (DNS) was introduced. In short, DNS is a distributed naming system that allows associating human-friendly names with devices in a computer network. A DNS name or a domain name is a string that represents a name of a device in the computer network.
To be precise, a DNS name is an alias for one or more IP addresses but not the devices. It doesn't name a particular physical device but an IP address that can be assigned to a device. Thus, DNS introduces a level of indirection in addressing a particular server application in the network.

DNS acts as a distributed database storing mappings of DNS names to corresponding IP addresses and providing an interface, allowing querying the IP addresses to which a particular DNS name is mapped. The process of transforming a DNS name into corresponding IP addresses is called a DNS name resolution. Modern network operating systems contain functionality that can query DNS to resolve DNS names and provides the interface that can be used by applications to perform DNS name resolution.

When given a DNS name, before a client can communicate with a corresponding server application, it must first resolve the name to obtain IP addresses associated with that name.

This recipe explains how to perform a DNS name resolution with Boost.Asio.

**How to do it...**

The following algorithm describes steps required to perform in a client application in order to resolve a DNS name to obtain IP addresses (zero or more) of hosts (zero or more) running the server application that the client application wants to communicate with:

1. Obtain the DNS name and the protocol port number designating the server application and represent them as strings.
2. Create an instance of the `asio::io_service` class or use the one that has been created earlier.
3. Create an object of the `resolver::query` class representing a DNS name resolution query.
4. Create an instance of DNS name resolver class suitable for the necessary protocol.
5. Call the resolver's `resolve()` method, passing a query object created in step 3 to it as an argument.

The following code sample demonstrates the possible implementation of the algorithm. It is assumed that the client application is intended to communicate with the server application over the TCP protocol and IPv6 as the underlying protocol. Besides, it is assumed that the server DNS name and a port number have already been obtained and represented as strings by the client application:

```cpp
#include <boost/asio.hpp>
#include <iostream>
```
```cpp
using namespace boost;

int main()
{
    // Step 1. Assume that the client application has already
    // obtained the DNS name and protocol port number and
    // represented them as strings.
    std::string host = "samplehost.com";
    std::string port_num = "3333";

    // Step 2.
    asio::io_service ios;

    // Step 3. Creating a query.
    asio::ip::tcp::resolver::query resolver_query(host,
        port_num, asio::ip::tcp::resolver::query::numeric_service);

    asio::ip::tcp::resolver resolver(ios);

    // Used to store information about error that happens
    // during the resolution process.
    boost::system::error_code ec;

    // Step 5.
    asio::ip::tcp::resolver::iterator it =
        resolver.resolve(resolver_query, ec);

    // Handling errors if any.
    if (ec != 0) {
        // Failed to resolve the DNS name. Breaking execution.
        std::cout << "Failed to resolve a DNS name."
        << "Error code = " << ec.value()
        << ". Message = " << ec.message();
        return ec.value();
    }

    return 0;
}
```
How it works...

In step 1, we begin by obtaining a DNS name and a protocol port number and representing them as strings. Usually, these parameters are supplied by a user through the client application’s UI or as command-line arguments. The process of obtaining and validating these parameters is behind the scope of this recipe; therefore, here we assume that they are available at the beginning of the sample.

Then, in step 2, we create an instance of the `asio::io_service` class that is used by the resolver to access underlying OS’s services during a DNS name resolution process.

In step 3 we create an object of the `asio::ip::tcp::resolver::query` class. This object represents a query to the DNS. It contains a DNS name to resolve, a port number that will be used to construct an endpoint object after the DNS name resolution and a set of flags controlling some specific aspects of resolution process, represented as a bitmap. All these values are passed to the query class’s constructor. Because the service is specified as a protocol port number (in our case, 3333) and not as a service name (for example, HTTP, FTP, and so on), we passed the `asio::ip::tcp::resolver::query::numeric_service` flag to explicitly inform the query object about that, so that it properly parses the port number value.

In step 4, we create an instance of the `asio::ip::tcp::resolver` class. This class provides the DNS name resolution functionality. To perform the resolution, it requires services of the underlying operating system and it gets access to them through the object of the `asio::io_services` class being passed to its constructor as an argument.

The DNS name resolution is performed in step 5 in the resolver object’s `resolve()` method. The method overload we use in our sample accepts objects of the `asio::ip::tcp::resolver::query` and `system::error_code` classes. The latter object will contain information describing the error if the method fails.

If successful, the method returns an object of the `asio::ip::tcp::resolver::iterator` class, which is an iterator pointing to the first element of a collection representing resolution results. The collection contains objects of the `asio::ip::basic_resolver_entry<tcp>` class. There are as many objects in the collection as the total number of IP addresses that resolution yielded. Each collection element contains an object of the `asio::ip::tcp::endpoint` class instantiated from one IP address resulting from the resolution process and a port number provided with the corresponding query object. The endpoint object can be accessed through the `asio::ip::basic_resolver_entry<tcp>::endpoint()` getter method.
The default-constructed object of the \texttt{asio::ip::tcp::resolver::iterator} class represents an end iterator. Consider the following sample demonstrating how we can iterate through the elements of the collection representing the DNS name resolution process results and how to access the resulting endpoint objects:

\begin{verbatim}
asio::ip::tcp::resolver::iterator it = resolver.resolve(resolver_query, ec);
asio::ip::tcp::resolver::iterator it_end;

for (; it != it_end; ++it) {
    // Here we can access the endpoint like this.
    asio::ip::tcp::endpoint ep = it->endpoint();
}
\end{verbatim}

Usually, when a DNS name of the host running the server application is resolved to more than one IP address and correspondingly to more than one endpoint, the client application doesn't know which one of the multiple endpoints to prefer. The common approach in this case is to try to communicate with each endpoint one by one, until the desired response is received.

Note that when the DNS name is mapped to more than one IP address and some of them are IPv4 and others are IPv6 addresses, the DNS name may be resolved either to the IPv4 address or to the IPv6 address or to both. Therefore, the resulting collection may contain endpoints representing both IPv4 and IPv6 addresses.

\textbf{There's more...}

To resolve a DNS name and obtain a collection of endpoints that can be used in the client that is intended to communicate over the UDP protocol, the code is very similar. The sample is given here with differences highlighted and without explanation:

\begin{verbatim}
#include <boost/asio.hpp>
#include <iostream>

using namespace boost;

int main()
{
    // Step 1. Assume that the client application has already
    // obtained the DNS name and protocol port number and
    // represented them as strings.
\end{verbatim}
std::string host = "samplehost.book";
std::string port_num = "3333";

// Step 2.
asio::io_service ios;

// Step 3. Creating a query.
asio::ip::udp::resolver::query resolver_query(host,
port_num, asio::ip::udp::resolver::query::numeric_service);

asio::ip::udp::resolver resolver(ios);

// Used to store information about error that happens
// during the resolution process.
boost::system::error_code ec;

// Step 5.
asio::ip::udp::resolver::iterator it =
  resolver.resolve(resolver_query, ec);

// Handling errors if any.
if (ec != 0) {
  // Failed to resolve the DNS name. Breaking execution.
  std::cout << "Failed to resolve a DNS name."
  << " Error code = " << ec.value()
  << ". Message = " << ec.message();

  return ec.value();
}

asio::ip::udp::resolver::iterator it_end;
for (; it != it_end; ++it) {
  // Here we can access the endpoint like this.
  asio::ip::udp::endpoint ep = it->endpoint();
}

return 0;
See also

- The Creating an endpoint recipe provides more information on endpoints
- For more information on DNS and domain names, refer to the specification of the system that can be found in the RFC #1034 and RFC #1035 documents

Binding a socket to an endpoint

Before an active socket can communicate with a remote application or a passive socket can accept incoming connection requests, they must be associated with a particular local IP address (or multiple addresses) and a protocol port number, that is, an endpoint. The process of associating a socket with a particular endpoint is called binding. When a socket is bound to an endpoint, all network packets coming into the host from the network with that endpoint as their target address will be redirected to that particular socket by the operating system. Likewise, all the data coming out from a socket bound to a particular endpoint will be output from the host to the network through a network interface associated with the corresponding IP address specified in that endpoint.

Some operations bind unbound sockets implicitly. For example, an operation that connects an unbound active socket to a remote application, binds it implicitly to an IP address and a protocol port number chosen by the underlying operating system. Usually, the client application doesn't need to explicitly bind an active socket to a specific endpoint just because it doesn't need that specific endpoint to communicate with the server; it only needs any endpoint for that purpose. Therefore, it usually delegates the right to choose the IP address and the port number to which the socket should be bound to the operating system. However, in some special cases, the client application might need to use a specific IP address and a protocol port number to communicate with the remote application and therefore will bind its socket explicitly to that specific endpoint. We wouldn't consider these cases in our book.

When socket binding is delegated to the operating system, there is no guarantee that it will be bound to the same endpoint each time. Even if there is a single network interface and a single IP address on the host, the socket may be bound to a different protocol port number every time the implicit binding is performed.

Unlike client applications that usually don't care through which IP address and protocol port number its active socket will be communicating with the remote application, the server application usually needs to bind its acceptor socket to a particular endpoint explicitly. This is explained by the fact that the server's endpoint must be known to all the clients that want to communicate with it and should stay the same after the server application is restarted.

This recipe explains how to bind a socket to particular endpoint with Boost.Asio.
How to do it...

The following algorithm describes steps required to create an acceptor socket and to bind it to an endpoint designating all IP addresses available on the host and a particular protocol port number in the IPv4 TCP server application:

1. Obtain the protocol port number on which the server should listen for incoming connection requests.
2. Create an endpoint that represents all IP addresses available on the host and the protocol port number obtained in the step 1.
3. Create and open an acceptor socket.
4. Call the acceptor socket’s `bind()` method, passing the endpoint object as an argument to it.

The following code sample demonstrates possible implementation of the algorithm. It is assumed that the protocol port number has already been obtained by the application:

```cpp
#include <boost/asio.hpp>
#include <iostream>
using namespace boost;

int main()
{
    // Step 1. Here we assume that the server application has
    // already obtained the protocol port number.
    unsigned short port_num = 3333;

    // Step 2. Creating an endpoint.
    asio::ip::tcp::endpoint ep(asio::ip::address_v4::any(),
                                port_num);

    // Used by 'acceptor' class constructor.
    asio::io_service ios;

    // Step 3. Creating and opening an acceptor socket.
    asio::ip::tcp::acceptor acceptor(ios, ep.protocol());

    boost::system::error_code ec;

    // Step 4. Binding the acceptor socket.
    acceptor.bind(ep, ec);
```
// Handling errors if any.
if (ec != 0) {
    // Failed to bind the acceptor socket. Breaking
    // execution.
    std::cout << "Failed to bind the acceptor socket."
    << "Error code = " << ec.value() << ". Message: "
    << ec.message();

    return ec.value();
}

return 0;

How it works...

We begin by obtaining a protocol port number in step 1. The process of obtaining this parameter is beyond the scope of this recipe; therefore, here we assume that the port number has already been obtained and is available at the beginning of the sample.

In step 2 we create an endpoint representing all IP addresses available on the host and the specified port number.

In step 3 we instantiate and open the acceptor socket. The endpoint we created in step 2 contains information about the transport protocol and the version of the underlying IP protocol (IPv4). Therefore, we don't need to create another object representing the protocol to pass it to the acceptor socket's constructor. Instead, we use the endpoint's protocol() method, which returns an object of the asio: :ip: :tcp class representing the corresponding protocols.

The binding is performed in step 4. This is quite a simple operation. We call the acceptor socket's bind() method, passing an object representing an endpoint to which the acceptor socket should be bound as an argument of the method. If the call succeeds, the acceptor socket is bound to the corresponding endpoint and is ready to start listening for incoming connection requests on that endpoint.

Download the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
UDP servers don't establish connections and use active sockets to wait for incoming requests. The process of binding an active socket is very similar to binding an acceptor socket. Here, we present a sample code demonstrating how to bind a UDP active socket to an endpoint designating all IP addresses available on the host and a particular protocol port number. The code is provided without explanation:

```cpp
#include <boost/asio.hpp>
#include <iostream>

using namespace boost;

int main()
{
    // Step 1. Here we assume that the server application has
    // already obtained the protocol port number.
    unsigned short port_num = 3333;

    // Step 2. Creating an endpoint.
    asio::ip::udp::endpoint ep(asio::ip::address_v4::any(),
        port_num);

    // Used by 'socket' class constructor.
    asio::io_service ios;

    // Step 3. Creating and opening a socket.
    asio::ip::udp::socket sock(ios, ep.protocol());

    boost::system::error_code ec;

    // Step 4. Binding the socket to an endpoint.
    sock.bind(ep, ec);

    // Handling errors if any.
    if (ec != 0)
    {
        // Failed to bind the socket. Breaking execution.
        std::cout << "Failed to bind the socket."
        << "Error code = " << ec.value() << ". Message: "
        << ec.message();
    }
}
```
return ec.value();
}

return 0;
}

See also

- The Creating an endpoint recipe provides more information on endpoints
- The Creating an active socket recipe provides more details about the asio::io_service and asio::ip::tcp classes and demonstrates how to create and open an active socket
- The Creating a passive socket recipe provides information about passive sockets and demonstrates how to create and open them

Connecting a socket

Before a TCP socket can be used to communicate with a remote application, it must establish a logical connection with it. According to the TCP protocol, the connection establishment process lies in exchanging of service messages between two applications, which, if succeeds, results in two applications being logically connected and ready for communication with each other.

Roughly, the connection establishment process looks like this. The client application, when it wants to communicate with the server application, creates and opens an active socket and issues a connect() command on it, specifying a target server application with an endpoint object. This leads to a connection establishment request message being sent to the server application over the network. The server application receives the request and creates an active socket on its side, marking it as connected to a specific client and replies back to the client with the message acknowledging that connection is successfully set up on the server side. Next, the client having received the acknowledgement from the server, marks its socket as connected to the server, and sends one more message to it acknowledging that the connection is successfully set up on the client side. When the server receives the acknowledgement message from the client, the logical connection between two applications is considered established.

The point-to-point communication model is assumed between two connected sockets. This means that if socket A is connected to socket B, both can only communicate with each other and cannot communicate with any other socket C. Before socket A can communicate with socket C, it must close the connection with socket B and establish a new connection with socket C.

This recipe explains how to synchronously connect a socket to a remote application with Boost.Asio.
The Basics

How to do it...

The following algorithm describes steps required to perform in the TCP client application to connect an active socket to the server application:

1. Obtain the target server application’s IP address and a protocol port number.
2. Create an object of the asio::ip::tcp::endpoint class from the IP address and the protocol port number obtained in step 1.
3. Create and open an active socket.
4. Call the socket’s connect() method specifying the endpoint object created in step 2 as an argument.
5. If the method succeeds, the socket is considered connected and can be used to send and receive data to and from the server.

The following code sample demonstrates a possible implementation of the algorithm:

```cpp
#include <boost/asio.hpp>
#include <iostream>
using namespace boost;

int main()
{
    // Step 1. Assume that the client application has already obtained the IP address and protocol port number of the target server.
    std::string raw_ip_address = "127.0.0.1";
    unsigned short port_num = 3333;
    try {
        // Step 2. Creating an endpoint designating a target server application.
        asio::ip::tcp::endpoint ep(asio::ip::address::from_string(raw_ip_address), port_num);
        asio::io_service ios;
        // Step 3. Creating and opening a socket.
        asio::ip::tcp::socket sock(ios, ep.protocol());
```
Chapter 1

sock.connect(ep);

// At this point socket 'sock' is connected to
// the server application and can be used
// to send data to or receive data from it.

// Overloads of asio::ip::address::from_string() and
// asio::ip::tcp::socket::connect() used here throw
// exceptions in case of error condition.
catch (system::system_error &e) {
  std::cout << "Error occurred! Error code = " << e.code()
           << ". Message: " << e.what();

  return e.code().value();
}

return 0;

How it works...

In step 1, we begin with obtaining the target server's IP address and a protocol port number. The process of obtaining these parameters is beyond the scope of this recipe; therefore, here we assume that they have already been obtained and are available at the beginning of our sample.

In step 2, we create an object of the asio::ip::tcp::endpoint class designating the target server application to which we are going to connect.

Then, in step 3 an active socket is instantiated and opened.

In step 4, we call the socket's connect() method, passing an endpoint object designating the target server to it as an argument. This function connects the socket to the server. The connection is performed synchronously, which means that the method blocks the caller thread until either the connection operation is established or an error occurs.

Note that we didn't bind the socket to any local endpoint before connecting it. This doesn't mean that the socket stays unbound. Before performing the connection establishment procedure, the socket's connect() method will bind the socket to the endpoint consisting of an IP address and a protocol port number chosen by the operating system.
Another thing to note in this sample is that we use an overload of the `connect()` method that throws an exception of the `boost::system::system_error` type if the operation fails, and so does overload of the `asio::ip::address::from_string()` static method we use in step 2. Therefore, both calls are enclosed in a `try` block. Both methods have overloads that don't throw exceptions and accept an object of the `boost::system::error_code` class, which is used to conduct error information to the caller in case the operation fails. However, in this case, using exceptions to handle errors makes code better structured.

**There's more...**

The previous code sample showed how to connect a socket to a specific server application designated by an endpoint when an IP address and a protocol port number are provided to the client application explicitly. However, sometimes the client application is provided with a DNS name that may be mapped to one or more IP addresses. In this case, we first need to resolve the DNS name using the `resolve()` method provided by the `asio::ip::tcp::resolver` class. This method resolves a DNS name, creates an object of the `asio::ip::tcp::endpoint` class from each IP address resulted from resolution, puts all endpoint objects in a collection, and returns an object of the `asio::ip::tcp::resolver::iterator` class, which is an iterator pointing to the first element in the collection.

When a DNS name resolves to multiple IP addresses, the client application—when deciding to which one to connect—usually has no reasons to prefer one IP address to any other. The common approach in this situation is to iterate through endpoints in the collection and try to connect to each of them one by one until the connection succeeds. Boost.Asio provides auxiliary functionality that implements this approach.

The free function `asio::connect()` accepts an active socket object and an object of the `asio::ip::tcp::resolver::iterator` class as input arguments, iterates over a collection of endpoints, and tries to connect the socket to each endpoint. The function stops iteration, and returns when it either successfully connects a socket to one of the endpoints or when it has tried all the endpoints and failed to connect the socket to all of them.

The following algorithm demonstrates steps required to connect a socket to a server application represented by a DNS name and a protocol port number:

1. Obtain the DNS name of a host running the server application and the server's port number and represent them as strings.
2. Resolve a DNS name using the `asio::ip::tcp::resolver` class.
3. Create an active socket without opening it.
4. Call the `asio::connect()` function passing a socket object and an iterator object obtained in step 2 to it as arguments.
Chapter 1

The following code sample demonstrates possible implementation of the algorithm:

```cpp
#include <boost/asio.hpp>
#include <iostream>

using namespace boost;

int main()
{
    // Step1. Assume that the client application has already
    // obtained the DNS name and protocol port number and
    // represented them as strings.
    std::string host = "samplehost.book";
    std::string port_num = "3333";

    // Used by a 'resolver' and a 'socket'.
    asio::io_service ios;

    // Creating a resolver's query.
    asio::ip::tcp::resolver::query resolver_query(host, port_num,
        asio::ip::tcp::resolver::query::numeric_service);

    // Creating a resolver.
    asio::ip::tcp::resolver resolver(ios);
    try {
        // Step 2. Resolving a DNS name.
        asio::ip::tcp::resolver::iterator it =
            resolver.resolve(resolver_query);

        // Step 3. Creating a socket.
        asio::ip::tcp::socket sock(ios);

        // Step 4. asio::connect() method iterates over
        // each endpoint until successfully connects to one
        // of them. It will throw an exception if it fails
        // to connect to all the endpoints or if other
        // error occurs.
        asio::connect(sock, it);

        // At this point socket 'sock' is connected to
        // the server application and can be used
        // to send data to or receive data from it.
    }
```
The Basics

```cpp
// Overloads of asio::ip::tcp::resolver::resolve and
// asio::connect() used here throw
// exceptions in case of error condition.
    catch (system::system_error &e) {
        std::cout << "Error occurred! Error code = " << e.code() << ". Message: " << e.what();

        return e.code().value();
    }

    return 0;
}
```

Note that in step 3, we don't open the socket when we create it. This is because we don't know the version of IP addresses to which the provided DNS name will resolve. The `asio::connect()` function opens the socket before connecting it to each endpoint specifying proper protocol object and closes it if the connection fails.

All other steps in the code sample should not be difficult to understand, therefore no explanation is provided.

**See also**

- The *Creating an endpoint* recipe provides more information on endpoints
- The *Creating an active socket* recipe explains how to create and open a socket and provides more details about the `asio::io_service` class
- The *Resolving a DNS name* recipe explains how to use a resolver class to resolve a DNS name
- The *Binding a socket* recipe provides more information about socket binding

**Accepting connections**

When the client application wants to communicate to the server application over a TCP protocol, it first needs to establish a logical connection with that server. In order to do that, the client allocates an active socket and issues a connect command on it (for example by calling the `connect()` method on the socket object), which leads to a connection establishment request message being sent to the server.

On the server side, some arrangements must be performed before the server application can accept and handle the connection requests arriving from the clients. Before that, all connection requests targeted at this server application are rejected by the operating system.
First, the server application creates and opens an acceptor socket and binds it to the particular endpoint. At this point, the client’s connection requests arriving at the acceptor socket's endpoint are still rejected by the operating system. For the operating system to start accepting connection requests targeted at particular endpoint associated with particular acceptor socket, that acceptor socket must be switched into listening mode. After that, the operating system allocates a queue for pending connection requests associated with this acceptor socket and starts accepting connection request addressed to it.

When a new connection request arrives, it is initially received by the operating system, which puts it to the pending connection requests queue associated with an acceptor socket being the connection request's target. When in the queue, the connection request is available to the server application for processing. The server application, when ready to process the next connection request, de-queues one and processes it.

Note that the acceptor socket is only used to establish connections with client applications and is not used in the further communication process. When processing a pending connection request, the acceptor socket allocates a new active socket, binds it to an endpoint chosen by the operating system, and connects it to the corresponding client application that has issued that connection request. Then, this new active socket is ready to be used for communication with the client. The acceptor socket becomes available to process the next pending connection request.

This recipe describes how to switch an acceptor socket into listening mode and accept incoming connection requests in a TCP server application using Boost.Asio.

How to do it...

The following algorithm describes how to set up an acceptor socket so that it starts listening for incoming connections and then how to use it to synchronously process the pending connection request. The algorithm assumes that only one incoming connection will be processed in synchronous mode:

1. Obtain the port number on which the server will receive incoming connection requests.
2. Create a server endpoint.
3. Instantiate and open an acceptor socket.
4. Bind the acceptor socket to the server endpoint created in step 2.
5. Call the acceptor socket’s listen() method to make it start listening for incoming connection requests on the endpoint.
6. Instantiate an active socket object.
7. When ready to process a connection request, call the acceptor socket’s `accept()` method passing an active socket object created in step 6 as an argument.

8. If the call succeeds, the active socket is connected to the client application and is ready to be used for communication with it.

The following code sample demonstrates possible implementation of the server application that follows the algorithm. Here, we assume that the server is intended to communicate over the TCP protocol with IPv4 as the underlying protocol:

```cpp
#include <boost/asio.hpp>
#include <iostream>

using namespace boost;

int main()
{
    // The size of the queue containing the pending connection requests.
    const int BACKLOG_SIZE = 30;

    // Step 1. Here we assume that the server application has already obtained the protocol port number.
    unsigned short port_num = 3333;

    // Step 2. Creating a server endpoint.
    asio::ip::tcp::endpoint ep(asio::ip::address_v4::any(), port_num);

    asio::io_service ios;

    try {
        // Step 3. Instantiating and opening an acceptor socket.
        asio::ip::tcp::acceptor acceptor(ios, ep.protocol());

        // Step 4. Binding the acceptor socket to the server endpoint.
        acceptor.bind(ep);

        // Step 5. Starting to listen for incoming connection requests.
        acceptor.listen(BACKLOG_SIZE);
    }
    catch (std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
}
```
asio::ip::tcp::socket sock(ios);

// Step 7. Processing the next connection request and
// connecting the active socket to the client.
acceptor.accept(sock);

// At this point 'sock' socket is connected to
// the client application and can be used to send data to
// or receive data from it.
}
catch (system::system_error &e) {
    std::cout << "Error occurred! Error code = " << e.code() << ". Message: " << e.what();

    return e.code().value();
}

return 0;

How it works...

In step 1, we obtain the protocol port number to which the server application binds its acceptor socket. Here, we assume that the port number has already been obtained and is available at the beginning of the sample.

In step 2, we create a server endpoint that designates all IP addresses available on the host running the server application and a specific protocol port number.

Then in step 3, we instantiate and open an acceptor socket and bind it to the server endpoint in step 4.

In step 5, we call the acceptor's listen() method passing the BACKLOG_SIZE constant value as an argument. This call switches the acceptor socket into the state in which it listens for incoming connection requests. Unless we call the listen() method on the acceptor object, all connection requests arriving at corresponding endpoint will be rejected by the operating system network software. The application must explicitly notify the operating system that it wants to start listening for incoming connection requests on specific endpoint by this call.
The argument that the `listen()` method accepts as an argument specifies the size of the queue maintained by the operating system to which it puts connection requests arriving from the clients. The requests stay in the queue and are waiting for the server application to dequeue and process them. When the queue becomes full, the new connection requests are rejected by the operating system.

In step 6, we create an active socket object without opening it. We'll need it in step 7.

In step 7, we call the acceptor socket's `accept()` method. This method accepts an active socket as an argument and performs several operations. First, it checks the queue associated with the acceptor socket containing pending connection requests. If the queue is empty, the method blocks execution until a new connection request arrives to an endpoint to which the acceptor socket is bound and the operating system puts it in the queue.

If at least one connection request is available in the queue, the one on the top of the queue is extracted from it and processed. The active socket that was passed to the `accept()` method as an argument is connected to the corresponding client application which issued the connection request.

If the connection establishment process succeeds, the `accept()` method returns and the active socket is opened and connected to the client application and can be used to send data to and receive data from it.

Remember that the acceptor socket doesn't connect itself to the client application while processing a connection request. Instead, it opens and connects another active socket, which is then used for communication with the client application. The acceptor socket only listens for and processes (accepts) incoming connection requests.

Note that UDP servers don't use acceptor sockets because the UDP protocol doesn't imply connection establishment. Instead, an active socket is used that is bound to an endpoint and listens for incoming I/O messages, and this same active socket is used for communication.

See also

- The Creating a passive socket recipe provides information about passive sockets and demonstrates how to create and open them
- The Creating an endpoint recipe provides more information on endpoints
- The Creating an active socket recipe explains how to create and open a socket and provides more details about the `asio::io_service` class
- The Binding a socket recipe provides more information about socket binding
Where to buy this book

You can buy Boost.Asio C++ Network Programming Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.