Chapter 4

Server Lifecycle

A server socket listens for connections rather than initiating them. The typical lifecycle looks something like this:

1. create
2. bind
3. listen
4. accept
5. close

We covered #1 in the previous chapter, now we'll continue on with the rest of the list.

Servers Bind

The second step in the lifecycle of a server socket is to **bind** to a port where it will listen for connections.

```ruby
./code/snippets/bind.rb
require 'socket'

# First, create a new TCP socket.
socket = Socket.new(:INET, :STREAM)

# Create a C struct to hold the address for listening.
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')

# Bind to it.
socket.bind(addr)
```

This is a low-level implementation that shows how to bind a TCP socket to a local port. In fact it's almost identical to the C source code you would write to accomplish the same thing.

This particular socket has now claimed port `4481` on the local host. Other sockets will not be able to bind to this port, doing so would result in an `Errno::EADDRINUSE` exception being raised. Client sockets will be able to connect to this socket using this port number, once a few more steps have been completed.

If you run that code block you'll notice that it exits immediately. The code works but doesn't yet do enough to actually listen for a connection. Keep reading to see how to put the server in `listen` mode.

So a server binds to a specific, agreed-upon port number which a client socket can then connect to.

Of course, Ruby provides syntactic sugar so that you never have to actually use `Socket.addr_in` or `Socket#bind` directly. But before learning the syntactic sugar it's important that we see how to do things the hard way.

What port should I bind to?

This is an important consideration for anyone writing a server. Should you pick a random port number? How can you tell if some other program has already 'claimed' a port as their own?

In terms of what's possible, any port from 1-65,535 *can* be used, but there are important conventions to consider before picking a port.

The first rule: **don't try to use a port in the 0-1024 range**. These are considered 'well-known' ports and are reserved for system use. A few examples: HTTP traffic happens on port 80, SMTP traffic happens on port 25, rsync happens on port 873. Binding to these ports typically requires root access.

The second rule: **don't use a port in the 49,000-65,535 range**. These are the ephemeral ports. They're typically used by services that don't operate on a predefined port number but need ports for temporary purposes. They're also an integral part of the connection negotiation process we'll see in the next section. Picking a port above this number might cause conflicts for some of your users.

Besides that, **any port from 1025-48,999 is fair game for your uses**. If you're planning on claiming one of those ports as *the* port for your server then you should have a look at the IANA list of registered ports [2] and make sure that it doesn't conflict with some other popular server program out there.

2. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt

Binding to Loopback Addresses

One thing to be wary of when developing your applications: if you bind to the loopback host address `127.0.0.1` then your server socket will only be able to handle connections that come from that exact same host.

If you want to be able to handle connections from *any* host then you should bind to the `0.0.0.0` address.

```ruby
./code/snippets/loopback_binding.rb
require 'socket'

# This socket will only be able to handle connections from
# clients on the 127.0.0.1 host.
local_socket = Socket.new(:INET, :STREAM)
local_addr = Socket.pack_sockaddr_in(4481, '127.0.0.1')
local_socket.bind(local_addr)

# This socket will be able to handle connections from any
# client on any host.
remote_socket = Socket.new(:INET, :STREAM)
remote_addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
remote_socket.bind(remote_addr)
```

Servers Listen

After creating a socket, and binding to a port, the socket needs to be told to listen for incoming connections.

```ruby
./code/snippets/listen.rb
require 'socket'

# Create a socket and bind it to port 4481.
socket = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
socket.bind(addr)

# Tell it to listen for incoming connections.
socket.listen(5)
```

The only addition to the code from the last chapter is a call to `listen` on the socket.

If you run that code snippet it still exits immediately. There's one more step in the lifecycle of a server socket required before it can process connections. That's covered in the next chapter. First, more about `listen`.

The Listen Queue

You may have noticed that we passed an integer argument to the `listen` method. This number represents the maximum number of pending connections your server socket is willing to tolerate. This list of pending connections is called **the listen queue**.

Let's say that your server is busy processing a client connection, when any new client connections arrive they'll be put into the listen queue. If a new client connection arrives and the listen queue is full then an instance of `Errno::ECONNREFUSED` will be raised for the client.

How big should the listen queue be?

OK, so the size of the listen queue looks a bit like a magic number. Why wouldn't we want to set that number to 10,000? Why would we ever want to refuse a connection? All good questions.

First, we should talk about limits. You can get the current maximum allowed listen queue size by inspecting `Socket::SOMAXCONN` at runtime. On my Mac this number is 128. So I'm not able to use a number larger than that. The root user is able to increase this limit at the system level for servers that need it.

Ultimately you don't want to have connections waiting in your listen queue because that means that users of your service are having to wait for their responses.

Let's say you're running a server and you're getting of reports of `Errno::ECONNREFUSED`. That's *not* an indication that you need to increase the size of your listen queue. That would be acceptable only as a temporary solution. Rather this would be an indication that you need more server instances or that you need a different architecture. See the last part of this book for tips on the second part.

So the size of the listen queue depends on your needs and your limits. Generally you don't want to be refusing connections, so you can set it to the maximum allowed queue size using `server.listen(Socket::SOMAXCONN)`.

Servers Accept

Finally we get to the part of the lifecycle where the server is actually able to handle an incoming connection. It does this with the `accept` method. Here's how we can create a listening socket and receive the first connection:

```ruby
./code/snippets/accept.rb
require 'socket'

# Create the server socket.
server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(Socket::SOMAXCONN)

# Accept a connection.
connection, _ = server.accept
```

Now if you run that code you'll notice that it *doesn't* return immediately! That's right, the `accept` method will block until a connection arrives. Let's give it one using netcat:

```
$ echo ohai | nc localhost 4481
```

When you run these snippets you should see the nc(1) program and the Ruby program exit successfully. It may not be the most epic finale ever, but it's proof that everything is connected and working properly. Congrats!

Accept is blocking

The `accept` call is a blocking call. It will block the current thread indefinitely until it receives a new connection.

Remember the listen queue we talked about in the last chapter? `accept` simply pops the next pending connection off of that queue. If none are available it waits for one to be pushed onto it.

Accept returns an Array

In the example above I assigned two values from one call to `accept`. The `accept` method actually returns an Array. The Array contains two elements: first, the connection, and second, an `Addrinfo` object. This represents the local address of that connection.

Addrinfo

`Addrinfo` is a Ruby class that represents a host and port number. It wraps up an endpoint representation nicely. You'll see it as part of the standard `Socket` interface in a few places.

You can construct one of these using something like `Addrinfo.tcp('localhost', 4481)`. Some useful methods are `#ip_address` and `#ip_port`. Have a look at `$ ri Addrinfo` for more.

Let's begin by taking a closer look at the connection and addr returned from `#accept`.

This was a sample of *Working With TCP Sockets*.

If you haven't already, visit http://workingwithtcpsockets.com/ to be notified when it's ready.

Jesse (@jstorimer)