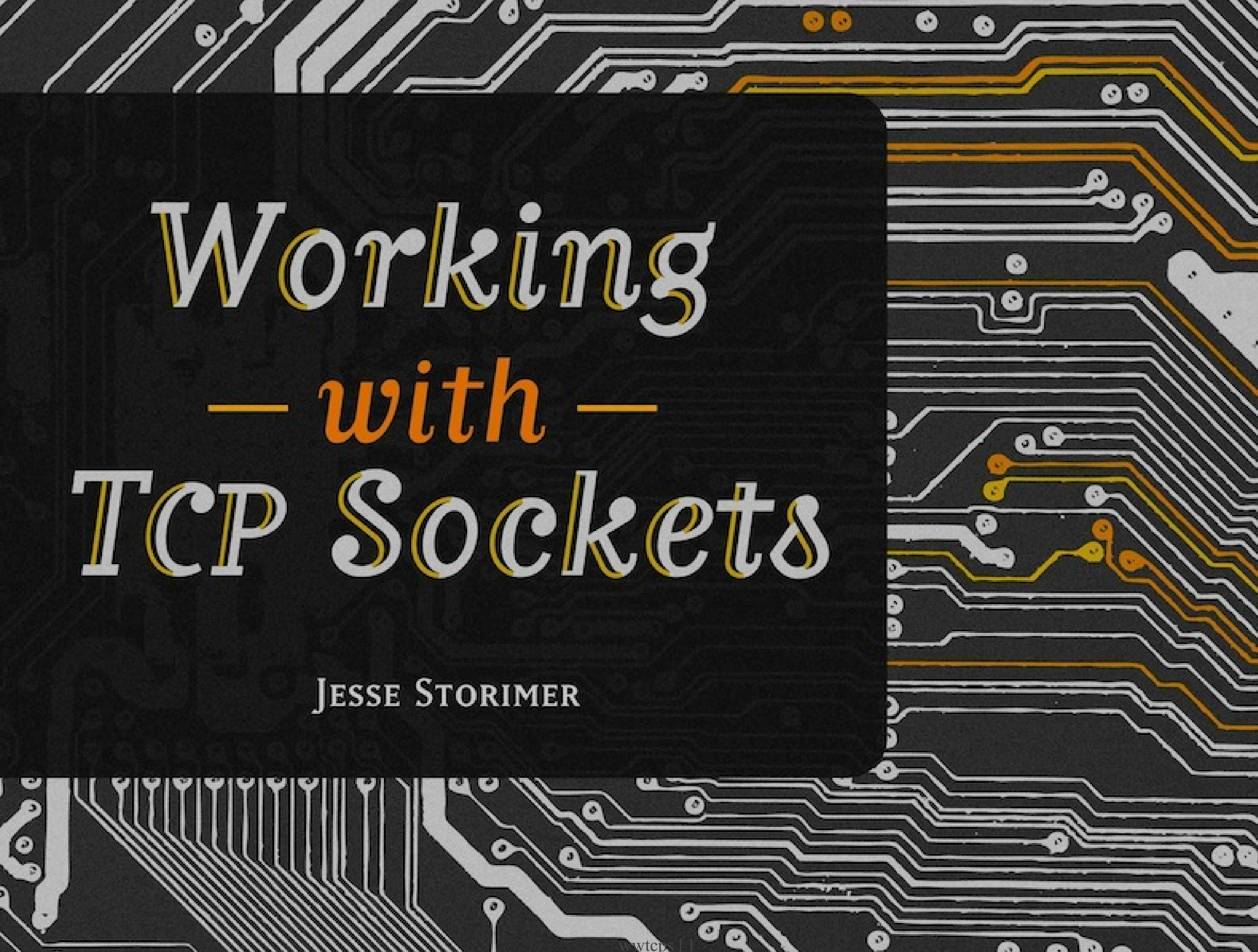

Working with TCP Sockets

Jesse Storimer

Contents

Releases

- October 24, 2012 - Initial public release.

Chapter 0

Introduction

Sockets connect the digital world.

Think for a minute about the early days of computing. Computers were something used exclusively by the scientists of the day. They were used for mathematical calculations, simulations; Real Serious Stuff™.

It was many years later when computers were able to connect people that the layperson became interested. Today, there are far more computers being used by laypeople than by scientists. Computers became interesting for this group when they could share information and communicate with anyone, anywhere.

It was network programming, and more specifically the proliferation of a particular socket programming API that allowed this to happen. Chances are, if you're reading this book, then you spend time every day connecting with people online and working with technology built on the idea of connecting computers together.

So network programming is ultimately about sharing and communication. This book exists so that you can better understand the underlying mechanisms of network programming and make better contributions to its cause.

My Story

I remember my first interaction with the world of sockets. It wasn't pretty.

As a web developer I had experience integrating with all kinds of HTTP APIs. I was accustomed to working with high-level concepts like REST & JSON.

Then I had to integrate with a domain registrar API.

I got a hold of the API documentation and was shocked. They wanted me to open a TCP socket on some private host name at some random port. This didn't work anything like the Twitter API!

Not only were they asking for a TCP socket, but they didn't encode data as JSON, or even XML. They had their own line protocol I had to adhere to. I had to send a very specifically formatted line of text over the socket, then send an empty line, then key-value pairs for the arguments, followed by two empty lines to show the request was done.

Then I had to read back a response in the same way. I was thinking "What in the...".

I showed this to a co-worker and he shared my trepidation. He had never worked with an API like this. He quickly warned me: "I've only ever used sockets in C. You have to be careful. Make sure you always close it before exiting otherwise it can stay open forever. They're hard to close once the program exits".

What?! Open forever? Protocols? Ports? I was flabbergasted.

Then another co-worker took a look and said "Really? You don't know how to work with sockets? You *do* know that you're opening a socket every time you read a web page, right? You should really know how this works."

I took that as a challenge. It was tough to wrap my head around the concepts at first, but I kept trying. I made lots of mistakes, but ultimately completed the integration. I think I'm a better programmer for it. It gave me a better understanding of the technology that my work depends upon. It's a good feeling.

With this book I hope to spare you some of that pain I felt when I was introduced to sockets, while still bringing you the sweet clarity that comes with having a deep understanding of your technology stack.

Who is This Book For?

The intended audience is Ruby developers on Unix or Unix-like systems.

The book assumes that you know Ruby and makes no attempts to teach Ruby basics. It assumes little to no knowledge of network programming concepts. It starts right at the fundamentals.

What to Expect

This book is divided into three main parts.

The first part gives an introduction to the primitives of socket programming. You'll learn how to create sockets, connect them together, and share data.

The second part of the book covers more advanced topics in socket programming. These are the kinds of things you'll need once you get past doing 'Hello world'-style socket programming.

The third part applies everything from the first two parts of the book in a 'real-world' scenario. This section goes past just sockets and shows you how to apply concurrency to your network programs. Several architecture patterns are implemented and compared to solve the same problem.

The Berkeley Sockets API

The main focus of this book will be the Berkeley Sockets API and its usage. The Berkeley Sockets API first appeared with version 4.2 of the BSD operating system in 1983. It was the first implementation of the then newly proposed Transport Control Protocol (TCP).

The Berkeley Sockets API has truly stood the test of time. The API that you'll work with in this book and the one supported in most modern programming languages is the same API that was revealed to the world in 1983.

Surely one key reason why the Berkeley Sockets API has stood the test of time: **You can use sockets without having to know the details of the underlying protocol**. This point is key and will get more attention later.

The Berkeley Sockets API is a programming API that operates at a level above the actually protocol implementation itself. It's concerned with stuff like connecting two endpoints and sharing data between them rather than marshalling packets and sequence numbering.

The de facto Berkeley Sockets API implementation is written in C, but almost any modern language written in C will include bindings to that lower-level interface. As such, there are many places in the book where I've gone to the effort of making the knowledge portable.

That is to say, rather than *just* showing the wrapper classes that Ruby offers around socket APIs I always start by showing the lower level API, followed by Ruby's wrapper classes. This keeps your knowledge portable.

When you're working in a language other than Ruby you'll still be able to apply the fundamentals you learn here and use the lower level constructs to build what you need.

What's Not Covered?

I mentioned in the last chapter that one of the strengths of the Berkeley Sockets API is that you don't need to know anything about the underlying protocol in order to use it. This book heartily embraces that.

Some other networking books focus on explaining the underlying protocol and its intricacies, even going as far as to re-implement TCP on top of another protocol like UDP or raw sockets. This book won't go there.

It will embrace the notion that the Berkeley Sockets API can be used without knowing the underlying protocol implementation. It will focus on how to use the API to do interesting things and will keep as much focus as possible on getting real work done.

However, there are times, when making performance optimizations, for example, when a lack of understanding of the underlying protocol will prevent you from using a feature properly. In these cases I'll yield and explain the necessary bits so that the concepts are understood.

Back to protocols. I've already said that TCP won't be covered in detail. The same is true for application protocols like HTTP, FTP, etc.. We'll look at some of these as examples, but not in detail.

If you're really interested in learning about the protocol itself I'd recommend Stevens' TCP/IP Illustrated [2].

netcat

There are several places in this book where the `netcat` tool is used to create arbitrary connections to test the various programs we're writing. `netcat` (usually `nc` in your terminal) is a Unix utility for creating arbitrary TCP (and UDP) connections and listens. It's a useful tool to have in your toolbox when working with sockets.

If you're on a Unix system it's likely already installed and you should have no issues with the examples.

Acknowledgements

First and foremost I have to thank my family: Sara and Inara. They didn't write the text, but they contributed in their own unique ways. From giving me the time and space to work on this, to reminding me what's important, if it weren't for them this book certainly wouldn't exist.

Next up are my awesome reviewers. These people read drafts of the book and together provided pages and pages of insights and comments that improved this book. Big thanks to Jonathan Rudenberg, Henrik Nyh, Cody Fauser, Julien Boyer, Joshua Wehner, Mike Perham, Camilo Lopez, Pat Shaughnessy, Trevor Bramble, Ryan LeCompte, Joe James, Michael Bernstein, Jesus Castello, and Pradeepto Bhattacharya.

[2] http://www.amazon.com/TCP-Illustrated-Vol-Addison-Wesley-Professional/dp/0201633469

Chapter 1

Your First Socket

Let's hit the ground running with an example.

Ruby's Socket Library

Ruby's Socket classes are not loaded by default. Everything that you need can be imported with `require 'socket'`. This includes a whole lot of different classes for TCP sockets, UDP sockets, as well as all the necessary primitives. You'll get a look at some of these throughout the book.

> The 'socket' library is part of Ruby's standard library. Similar to 'openssl', 'zlib', or 'curses', the 'socket' library provides thin bindings to dependable C libraries that have remained stable over many releases of Ruby.

So don't forget to `require 'socket'` before trying to create a socket.

Creating Your First Socket

On that note, let's dive in and create a socket:

```ruby
# ./code/snippets/create_socket.rb
require 'socket'

socket = Socket.new(Socket::AF_INET, Socket::SOCK_STREAM)
```

This creates a socket of type `STREAM` in the `INET` domain. The `INET` is short for internet and specifically refers to a socket in the IPv4 family of protocols.

The `STREAM` part says you'll be communicating using a stream. This is provided by TCP. If you had said `DGRAM` (for datagram) instead of `STREAM` that would refer to a UDP socket. The type tells the kernel what kind of socket to create.

Understanding Endpoints

I just threw out some new language there in talking about IPv4. Let's understand IPv4 and addressing before continuing.

When there are two sockets that want to communicate, they need to know where to find each other. This works much like a phone call: if you want to have a phone conversation with someone then you need to know their phone number.

Sockets use IP addresses to route messages to specific hosts. A host is identified by a unique IP address. This is its 'phone number'.

Above I specifically mentioned IPv4 addresses. An IPv4 address typically looks something like this: `192.168.0.1`. It's four numbers <= 255 joined with dots. What does that do? Armed with an IP address one host is able to route data to another host at that specific IP address.

The IP Address Phone Book

It's easy enough to imagine socket communication when you know the address of the host you want to communicate with, but how does one get that address? Does it need to be memorized? Written down? Thankfully no.

You've likely heard of DNS before. This is a system that maps host names to IP addresses. In this way you don't need to remember the specific address of the host you want to talk to, but you do need to remember its name. Then you can ask DNS to resolve that name to an address. Even if the underlying address changes, the host name will always get you to the right place. Bingo.

Loopbacks

IP addresses don't always have to refer to remote hosts. Especially in development you often want to connect to sockets on your local host.

Most systems define a loopback interface. This is an entirely virtual interface and, unlike the interface to your network card, is not attached to any hardware. Any data sent to the loopback interface is immediately received on the same interface. With a loopback address your network is constrained to the local host.

The host name for the loopback interface is officially called `localhost` and the loopback IP address is typically `127.0.0.1`. These are defined in a 'hosts' file for your system.

IPv6

I've mentioned IPv4 a few times, but have neglected to mention IPv6. IPv6 is an alternative addressing scheme for IP addresses.

Why does it exist? Because we literally ran out of IPv4 addresses. IPv4 consists of four numbers each in the range of 0-255. Each of these four numbers can be represented with 8 bits, giving us 32 bits total in the address. That means there are 2^{32} or 4.3 billion possible addresses. This a large number, but you can imagine how many individual devices are connected to networks that you see every day... it's no wonder we're running out.

So IPv6 is a bit of an elephant in the room at the moment. With IPv4 addresses now being exhausted [2], IPv6 is necessarily becoming relevant. It has a different format that allows for an astronomical number of unique IP addresses.

But for the most part you don't need to be typing these things out by hand and the interaction with either addressing scheme will be identical.

Ports

There's one more aspect that's crucial to an endpoint: the port number. Continuing with our phone call example: if you want to have a conversation with someone in an office building you'll have to call their phone number, then dial their extension. The port number is the 'extension' of a socket endpoint.

[2] http://www.nro.net/news/ipv4-free-pool-depleted

Each socket must bind to a unique port number. Without ports, a host would only be able to support one socket at a time. By marrying each active socket to a specific port number a host is able to support thousands of sockets concurrently.

Which port number should I use?

This problem is solved not with DNS, but with a list of well-defined port numbers.

For example, HTTP communication happens on port 80 by default, FTP communication on port 21. There is actually an organization [3] responsible for maintaining this list. More on port numbers in the next chapter.

Creating Your Second Socket

Now we get to see the first bit of syntactic sugar that Ruby offers.

Although there are much higher-level abstractions than this for creating sockets Ruby also lets you represent the different options as symbols instead of constants. So `Socket::AF_INET` becomes `:INET` and `Socket::SOCK_STREAM` becomes `:STREAM`. Here's an example of creating a TCP socket in the IPv6 domain:

[3] http://www.iana.org/

```ruby
# ./code/snippets/create_socket_memoized.rb
require 'socket'

socket = Socket.new(:INET6, :STREAM)
```

This creates a socket, but it's not yet ready to exchange data with other sockets. The next chapter will look at taking a socket like this and preparing it to do actual work.

Docs

Now seems like a good time to bring up documentation. One nice thing about doing socket programming is that you already have lots of documentation on your machine that can help you out. There are two primary places to find documentation for this stuff: 1) manpages, and 2) ri.

Let's do a quick review of each in turn.

1. *Unix Manual pages* will provide documentation about underlying system functions (C code). These are the primitives that Ruby's socket library is built upon. The manpages are thorough, but very low-level. They can give you an idea of what a system call does where Ruby's docs are lacking. It can also tell you what error codes are possible.

   For example, in the code sample above we used `Socket.new`. This maps to a system function called `socket()` which creates a socket. We can see the manpage for this using the following command:

```
$ man 2 socket
```

Notice the `2`? This tells the `man` program to look in section 2 of the manpages. The entire set of manpages is divided into sections.

- Section 1 is for 'General Commands' (shell programs)
- Section 2 is for system calls
- Section 3 is for C library functions
- Section 4 is for 'Special Files'
- Section 5 is for 'File Formats'
- Section 7 provides overviews on various topic. tcp(7) is of interest.

I'll refer to manpages using this syntax: socket(2). This refers to the socket manpage in section 2. This is necessary because some manpages exist in multiple sections. Take stat(1) and stat(2) as an example.

If you have a look at the 'SEE ALSO' section of socket(2), you'll see some of the other system calls we'll be looking at.

2. *ri* is Ruby's command line documentation tool. The Ruby installer installs documentation for the core library as part of the installation process.

   Some parts of Ruby aren't very well documented, but I must say that the socket library is pretty well covered. Let's look at the `ri` docs for `Socket.new`. We use the following command:

```
$ ri Socket.new
```

ri is useful and doesn't require an internet connection. It's a good place to look if you need guidance or examples.

System Calls From This Chapter

Each chapter will list any new system calls that were introduced and show you were you can find out more about them using `ri` or manpages.

- `Socket.new` -> socket(2).

Chapter 2

Establishing Connections

TCP connections are made between two endpoints. The endpoints may be on the same machine or on machines in two different parts of the world. Yet the principles behind each are the same.

When you create a socket it must assume one of two roles: 1) initiator, or 2) listener. Both roles are required. Without a listener socket no connection can be initiated. Similarly, without an initiator there's no need for a listener.

In network programming the term commonly used for **a socket that listens is a server** and **a socket that initiates a connection is a client**. We'll look at the lifecycle of each in turn.

Chapter 3

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 already; 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 code you would write to accomplish the same thing.

This particular socket is now bound to 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.

To recap, 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.pack_sockaddr_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 defaults to port 80, SMTP traffic defaults to port 25, rsync defaults to 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 in this range might cause issues 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 your choice doesn't conflict with some other popular server out there.

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

What address should I bind to?

I bound to `0.0.0.0` in the above example, but what's the difference when I bind to `127.0.0.1`? Or `1.2.3.4`? The answer has to do with interfaces.

Earlier I mentioned that your system has a loopback interface represented with the IP address `127.0.0.1`. It also has a physical, hardware-backed interface represented by a different IP address (let's pretend it's `192.168.0.5`). When you `bind` to a specific interface, represented by its IP address, your socket is only listening on that interface. It will ignore the others.

If you bind to `127.0.0.1` then your socket will only be listening on the loopback interface. In this case, only connections made to `localhost` or `127.0.0.1` will be routed to your server socket. Since this interface is only available locally, no external connections will be allowed.

If you bind to `192.168.0.5`, in this example, then your socket will only be listening on that interface. Any clients that can address that interface will be listened for, but any connections made on `localhost` will not be routed to that server socket.

If you want to listen on *all* interfaces then you can use `0.0.0.0`. This will bind to any available interface, loopback or otherwise. Most of the time, this is what you want.

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

# This socket will bind to the loopback interface and will
# only be listening for clients from localhost.
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 bind to any of the known interfaces and
# will be listening for any client that can route messages
# to it.
any_socket = Socket.new(:INET, :STREAM)
any_addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
any_socket.bind(any_addr)

# This socket attempts to bind to an unkown interface
# and raises Errno::EADDRNOTAVAIL.
error_socket = Socket.new(:INET, :STREAM)
error_addr = Socket.pack_sockaddr_in(4481, '1.2.3.4')
error_socket.bind(error_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 the client will raise `Errno::ECONNREFUSED`.

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.

Let's say you're running a server and you're getting of reports of `Errno::ECONNREFUSED`. Increasing the size of the listen queue would be a good starting point. But ultimately you don't want to have connections waiting in your listen queue. That means that users of your service are having to wait for their responses. This may be an indication that you need more server instances or that you need a different architecture.

Generally you don't want to be refusing connections. You can set the listen queue 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 to 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 remote address of the client 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 address returned from `#accept`.

```ruby
# ./code/snippets/accept_connection_class.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 new connection.
connection, _ = server.accept

print 'Connection class: '
p connection.class

print 'Server fileno: '
p server.fileno

print 'Connection fileno: '
p connection.fileno

print 'Local address: '
p connection.local_address

print 'Remote address: '
p connection.remote_address
```

When the server gets a connection (using the netcat snippet from above) it outputs:

```
Connection class: Socket
Server fileno: 5
Connection fileno: 8
Local address: #<Addrinfo: 127.0.0.1:4481 TCP>
Remote address: #<Addrinfo: 127.0.0.1:58164 TCP>
```

The results from this little bit of code tell us *a ton* about how TCP connections are handled. Let's dissect it a little bit at a time.

Connection Class

Although `accept` returns a 'connection', this code tells us that there's no special connection class. A connection is actually an instance of `Socket`.

File Descriptors

We know that `accept` is returning an instance of `Socket`, but this connection has a different file descriptor number (or `fileno`) than the server socket. The file descriptor number is the kernel's method for keeping track of open files in the current process.

Sockets are Files?

Yep. At least in the land of Unix everything is treated as a file[3]. This includes files found on the filesystem as well as things like pipes, sockets, printers, etc.

This indicates that `accept` has returned a brand new `Socket` different from the server socket. This `Socket` instance represents the connection. This is important. Each connection is represented by a new `Socket` object so that the server socket can remain untouched and continue to accept new connections.

Connection Addresses

Our connection object knows about two addresses: the local address and the remote address. The remote address is the second return value returned from `accept` but can also be accessed as `remote_address` on the connection.

The `local_address` of the connection refers to the endpoint on the local machine. The `remote_address` of the connection refers to the endpoint at the other end, which might be on another host but, in our case, it's on the same machine.

Each TCP connection is defined by this unique grouping of local-host, local-port, remote-host, and remote-port. The combination of these four properties *must* be unique for each TCP connection.

Let's put that in perspective for a moment. You can initiate two simultaneous connections from the local host to a remote host, so long as the remote ports are unique. Similarly you can accept two simultaneous connections from a remote host to the same local port, provided that the remote ports are unique. But you *cannot* have two simultaneous connections to a remote host if the local ports *and* remote ports are identical.

[3] http://ph7spot.com/musings/in-unix-everything-is-a-file

The Accept Loop

So `accept` returns one connection. In our code examples above the server accepts one connection and then exits. When writing a production server it's almost certain that we'd want to continually listen for incoming connections so long as there are more available. This is easily accomplished with a loop:

```
# ./code/snippets/naive_accept_loop.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)

# Enter an endless loop of accepting and
# handling connections.
loop do
  connection, _ = server.accept
  # handle connection
  connection.close
end
```

This is a common way to write certain kinds of servers using Ruby. It's so common in fact that Ruby provides some syntactic sugar on top of it. We'll look at Ruby wrapper methods at the end of this chapter.

Servers Close

Once a server has accepted a connection and finished processing it, the last thing for it to do is to `close` that connection. This rounds out the create-process-close lifecycle of a connection.

Rather than paste another block of code, I'll refer you to the one above. It calls `close` on the connection before accepting a new one.

Closing on Exit

Why is `close` needed? When your program exits, all open file descriptors (including sockets) will be closed for you. So why should you close them yourself? There are a few good reasons:

1. Resource usage. If you're done with a socket, but you don't close it, it's possible to store references to sockets you're no longer using. In Ruby's case the garbage collector is your friend, cleaning up any unreferenced connections for you, but it's a good idea to maintain full control over your resource usage and get rid of stuff you don't need. Note that the garbage collector will close anything that it collects.

2. Open file limit. This is really an extension of the previous one. Every process is subject to a limit on the number of open files it can have. Remember that each connection is a file? Keeping around unneeded connections will continue to bring your process closer and closer to this open file limit, which may cause issues later.

To find out the allowed number of open files for the current process you can use `Process.getrlimit(:NOFILE)`. The returned value is an Array of the soft limit (user-configurable) and hard limit (system), respectively.

If you want to bump up your limit to the maximum then you can `Process.setrlimit(Process.getrlimit(:NOFILE)[1])`.

Different Kinds of Closing

Given that sockets allow two-way communication (read/write) it's actually possible to close just one of those channels.

```ruby
# ./code/snippets/close_write.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)
connection, _ = server.accept

# After this the connection may no longer write data, but may still read data.
connection.close_write

# After this the connection may no longer read or write any data.
connection.close_read
```

Closing the write stream will send an `EOF` to the other end of the socket (more on EOF soon).

The `close_write` and `close_read` methods make use of shutdown(2) under the hood. shutdown(2) is notably different than close(2) in that it causes a part of the connection to be fully shut down, even if there are copies of it lying around.

How are there copies of connections?

It's possible to create copies of file descriptors using `Socket#dup`. This will actually duplicate the underlying file descriptor at the operating system level using dup(2). But this is pretty uncommon, and you probably won't see it.

The more common way that you can get a copy of a file descriptor is through `Process.fork`. This method creates a brand new process (Unix only) that's an exact copy of the current process. Besides providing a copy of everything in memory, any open file descriptors are dup(2)ed so that the new process gets a copy.

`close` will close the socket instance on which it's called. If there are other copies of the socket in the system then those *will not* be closed and the underlying resources will not be reclaimed. Indeed, other copies of the connection may still exchange data even if one instance is closed.

So `shutdown`, unlike `close`, will fully shut down communication on the current socket *and* other copies of it, thereby disabling any communication happening on the current instance as well as any copies. But it does not reclaim resources used by the socket. Each individual socket instance must still be `close`d to complete the lifecycle.

```ruby
# ./code/snippets/shutdown.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)
connection, _ = server.accept

# Create a copy of the connection.
copy = connection.dup

# This shuts down communication on all copies of the connection.
connection.shutdown

# This closes the original connection. The copy will be closed
# when the GC collects it.
connection.close
```

Ruby Wrappers

We all know and love the elegant syntax that Ruby offers, and its extensions for creating and working with server sockets are no exception. These convenience methods wrap up the boilerplate code in custom classes and leverage Ruby blocks where possible. Let's have a look.

Server Construction

First up is the `TCPServer` class. It's a clean way to abstract the 'server construction' part of the process.

```ruby
# ./code/snippets/server_easy_way.rb
require 'socket'

server = TCPServer.new(4481)
```

Ah, now *that* feels more like Ruby code. That code is effectively replacing this:

```ruby
# ./code/snippets/server_hard_way.rb
require 'socket'

server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(5)
```

I know which one I prefer to use!

> Creating a `TCPServer` instance actually returns an instance of `TCPServer`, *not* `Socket`. The interface exposed by each of them is nearly identical, but with some key differences. The most notable of which is that `TCPServer#accept` returns only the connection, not the `remote_address`.
>
> Notice that we didn't specify the size of the listen queue for these constructors? Rather than using `Socket::SOMAXCONN`, Ruby defaults to a listen queue of size `5`. If you need a bigger listen queue you can call `TCPServer#listen` after the fact.

As IPv6 gains momentum, your servers may need to be able to handle both IPv4 *and* IPv6. Using this Ruby wrapper will return two TCP sockets, one that can be reached via IPv4 and one that can be reached via IPv6, both listening on the same port.

```ruby
# ./code/snippets/server_sockets.rb
require 'socket'

servers = Socket.tcp_server_sockets(4481)
```

Connection Handling

Besides constructing servers, Ruby also provides nice abstractions for handling connections.

Remember using `loop` to handle multiple connections? Using `loop` is for chumps. Do it like this:

```ruby
# ./code/snippets/accept_loop.rb
require 'socket'

# Create the listener socket.
server = TCPServer.new(4481)

# Enter an endless loop of accepting and
# handling connections.
Socket.accept_loop(server) do |connection|
  # handle connection
  connection.close
end
```

Note that connections are *not* automatically closed at the end of each block. The arguments that get passed into the block are the exact same ones that are returned from a call to `accept`.

`Socket.accept_loop` has the added benefit that you can actually pass multiple listening sockets to it and it will accept connections on *any of the passed-in sockets*. This goes really well with `Socket.tcp_server_sockets`:

```
                                          # ./code/snippets/accept_server_sockets.rb
require 'socket'

# Create the listener socket.
servers = Socket.tcp_server_sockets(4481)

# Enter an endless loop of accepting and
# handling connections.
Socket.accept_loop(servers) do |connection|
  # handle connection
  connection.close
end
```

Notice that we're passing a collection of sockets to `Socket.accept_loop` and it handles them gracefully.

Wrapping it all into one

The granddaddy of the Ruby wrappers is `Socket.tcp_server_loop`, it wraps all of the previous steps into one:

```
                                                # ./code/snippets/tcp_server_loop.rb
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # handle connection
  connection.close
end
```

This method is really just a wrapper around `Socket.tcp_server_sockets` and `Socket.accept_loop`, but you can't write it any more succinctly than that!

System Calls From This Chapter

- `Socket#bind` -> bind(2)
- `Socket#listen` -> listen(2)
- `Socket#accept` -> accept(2)
- `Socket#local_address` -> getsockname(2)
- `Socket#remote_address` -> getpeername(2)
- `Socket#close` -> close(2)
- `Socket#close_write` -> shutdown(2)
- `Socket#shutdown` -> shutdown(2)

Chapter 4

Client Lifecycle

I mentioned that there are two critical roles that make up a network connection. The server takes the listening role, listening for and processing incoming connections. The client, on the other hand, takes the role of initiating those connections with the server. In other words, it knows the location of a particular server and creates an outbound connection to it.

As I'm sure is obvious, no server is complete without a client.

The client lifecycle is a bit shorter than that of the server. It looks something like this:

1. create
2. bind
3. connect
4. close

Step #1 is the same for both clients and servers, so we'll begin by looking at `bind` from the perspective of clients.

Clients Bind

Client sockets begin life in the same way as server sockets, with `bind`. In the server section we called `bind` with a specific address and port. While it's rare for a server to omit its call to `#bind`, **its rare for a client to make a call to bind**. If the client socket (or the server socket, for that matter) omit its call to bind, it will be assigned a random port from the ephemeral range.

Why not call bind?

Clients don't call `bind` because they don't need to be accessible from a known port number. The reason that servers bind to a specific port number is that clients expect a server to be available at a certain port number.

Take FTP as an example. The well-known port for FTP is 21. Hence FTP servers should bind to that port so that clients know where to find them. But the client is able to connect from any port number. The client port number does not affect the server.

Clients don't call `bind` because no one needs to know what their port number is.

There's no code snippet for this section because the recommendation is: don't do it!

Clients Connect

What really separates a client from a server is the call to `connect`. This call initiates a connection to a remote socket.

```ruby
# ./code/snippets/connect.rb
require 'socket'

socket = Socket.new(:INET, :STREAM)

# Initiate a connection to google.com on port 80.
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')
socket.connect(remote_addr)
```

Again, since we're using the low-level primitives here we're needing to pack the address object into its C struct representation.

This code snippet will initiate a TCP connection from a local port in the ephemeral range to a listening socket on port 80 of google.com. Notice that we didn't use a call to `bind`.

Connect Gone Awry

In the lifecycle of a client it's quite possible for a client socket to connect to a server before said server is ready to accept connections. It's equally possible to connect to a non-existent server. In fact, both of these situations produce the same outcome. Since TCP is optimistic, it waits as long as it can for a response from a remote host.

So, let's try connecting to an endpoint that's not available:

```
                                        # ./code/snippets/connect_non_existent.rb
require 'socket'

socket = Socket.new(:INET, :STREAM)

# Attempt to connect to google.com on the known gopher port.
remote_addr = Socket.pack_sockaddr_in(70, 'google.com')
socket.connect(remote_addr)
```

If you run this bit of code it can take a *long* time to return from the `connect` call. There is a long timeout by default on a `connect`.

This makes sense for clients where bandwidth is an issue and it may actually take a long time to establish a connection. Even for bandwidth-rich clients the default behaviour is hopeful that the remote address will be able to accept our connection soon.

Nevertheless, if you wait it out, you'll eventually see an `Errno::ETIMEDOUT` exception raised. This is the generic timeout exception when working with sockets and indicates that the requested operation timed out. If you're interested in tuning your socket timeouts there's an entire *Timeouts* chapter later in the book.

This same behaviour is observed when a client connects to a server that has called `bind` and `listen` but has not yet called `accept`. The only case in which `connect` returns successfully is if the remote server accepts the connect.

Ruby Wrappers

Much like the code for creating server sockets, the code for creating client sockets is verbose and low-level. As expected, Ruby has wrappers to make these easier to work with.

Client Construction

Before I show you the nice, rosy Ruby-friendly code I'm going to show the low-level verbose code so we have something to compare against:

```
# ./code/snippets/connect.rb
require 'socket'

socket = Socket.new(:INET, :STREAM)

# Initiate a connection to google.com on port 80.
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')
socket.connect(remote_addr)
```

Behold the syntactic sugar:

```
# ./code/snippets/client_easy_way.rb
require 'socket'

socket = TCPSocket.new('google.com', 80)
```

That feels much better. Three lines, two constructors, and lots of context have been reduced into a single constructor.

There's a similar client construction method using `Socket.tcp` that can take a block form:

```ruby
# ./code/snippets/client_block_form.rb
require 'socket'

Socket.tcp('google.com', 80) do |connection|
  connection.write "GET / HTTP/1.1\r\n"
  connection.close
end

# Omitting the block argument behaves the same
# as TCPSocket.new().
client = Socket.tcp('google.com', 80)
```

System Calls From This Chapter

- `Socket#bind` -> bind(2)
- `Socket#connect` -> connect(2)

Chapter 5

Exchanging Data

The previous section was all about establishing connections, connecting two endpoints together. While interesting in itself, you can't actually *do* anything interesting without exchanging data over a connection. This section gets into that. By the end we'll actually be able to wire up a server and client and have them talking to each other!

Before we dive in I'll just stress that it can be very helpful to think of a TCP connection as a series of tubes connecting a local socket to a remote socket, along which we can send and receive chunks of data. The Berkeley Sockets API was designed such that that we could model the world like this and have everything work out.

In the real world all of the data is encoded as TCP/IP packets and may visit many routers and hosts on the way to its destination. It's a bit of a crazy world and it's good to keep that in mind when things aren't working out, but thankfully, that crazy world is one that a lot of people worked very hard to cover up so we can stick to our simple mental model.

Streams

One more thing I need to drive home: the stream-based nature of TCP, something we haven't talked about yet.

Way back at the start of the book when we created our first socket we passed an option called `:STREAM` which said that we wanted to use a stream socket. TCP is a stream-based protocol. If we had not passed the `:STREAM` option when creating our socket it simply wouldn't be a TCP socket.

So what does that mean exactly? How does it affect the code?

First, I hinted at the term packets above. At the underlying protocol level TCP sends packets over the network.

But we're not going to talk about packets. From the perspective of your application code a TCP connection provides an ordered stream of communication with no beginning and no end. There is only the stream.

Let's illustrate this with some pseudo-code examples.

```
# This code sends three pieces of data over the network, one at a time.
data = ['a', 'b', 'c']

for piece in data
  write_to_connection(piece)
end

# This code consumes those three pieces of data in one operation.
result = read_from_connection #=> ['a', 'b', 'c']
```

The moral of the story here is that *a stream has no concept of message boundaries*. Even though the client sent three separate pieces of data, the server received them as one piece of data when it read them. It had no knowledge of the fact that the client sent the data in three distinct chunks.

Note that, although the message boundaries weren't preserved, the order of the content on the stream was preserved.

Chapter 6

Sockets Can Read

Thus far we've talked a lot about connections. Now we get to the really interesting part: how to pass data across socket connections. Unsurprisingly there is more than one way to read/write data when working with sockets, and on top of that, Ruby provides nice convenience wrappers for us.

This chapter will dive in to the different ways of reading data and when they're appropriate.

Simple Reads

The simplest way to read data from a socket is using the `read` method:

```ruby
# ./code/snippets/read.rb
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # Simplest way to read data from the connection.
  puts connection.read

  # Close the connection once we're done reading. Lets the client
  # know that they can stop waiting for us to write something back.
  connection.close
end
```

If you run that example in one terminal and the following `netcat` command in another terminal, you should see the output at the Ruby server this time:

```
$ echo gekko | nc localhost 4481
```

> If you've worked with Ruby's `File` API then this code may look familiar. Ruby's various socket classes, along with `File`, share a common parent in `IO`. All IO objects in Ruby (sockets, pipes, files, etc.) share a common interface supporting methods like `read`, `write`, `flush`, etc.
>
> Indeed, this isn't an innovation on Ruby's part. The underlying read(2), write(2), etc. system calls all function similarly with files, sockets, pipes, etc. This abstraction is built right into the core of the operating system itself. Remember, *everything is a file*.

It's Never That Simple

This method of reading data is simple, but brittle. If you run the example code again against this `netcat` command and leave it alone, the server will never finish reading the data and never exit:

```
$ tail -f /var/log/system.log | nc localhost 4481
```

The reason for this behaviour is something called EOF (end-of-file). It's covered in detail in the next section. For now we'll just play naive and look at the naive fix.

The gist of the issue is that `tail -f` never finishes sending data. If there is no data left to tail, it waits until there is some. Since it leaves its pipe open to `netcat`, then `netcat` too will never finish sending data to the server.

The server's call to `read` will continue blocking until the client finishes sending data. In this case the server will wait...and wait...and wait... meanwhile it's buffering whatever data it does receive in memory and not returning it to your program.

Read Length

One way around the above issue is to specify a minimum length to be read. That way, instead of continuing to read data until the client is finished you can tell the server to `read` a certain amount of data, then return.

```ruby
# ./code/snippets/read_with_length.rb
require 'socket'
one_kb = 1024 # bytes

Socket.tcp_server_loop(4481) do |connection|
  # Read data in chunks of 1 kb.
  while data = connection.read(one_kb) do
    puts data
  end

  connection.close
end
```

This above example, when run along with the same command:

```
$ tail -f /var/log/system.log | nc localhost 4481
```

will actually have the server printing data while the `netcat` command is still running. The data will be printed in one-kilobyte chunks.

The difference in this example is that we passed an integer to `read`. This tells it to stop reading and return what it has only once it has read that amount of data. Since we still want to get all the data available, we just loop over that `read` method calling it until it doesn't return any more data.

Blocking Nature

A call to `read` will always want to block and wait for the full length of data to arrive. Take our above example of reading one kilobyte at a time. After running it a few times, it should be obvious that if some amount of data has been read, but if that amount is less than one kilobyte, then `read` will continue to block until one full kilobyte can be returned.

It's actually possible to get yourself into a deadlock situation using this method. If a server attempts to `read` one kilobyte from the connection while the client sends only 500 bytes and then waits, the server will continue waiting for that full kilobyte!

This can be remedied in two ways: 1) the client sends an EOF after sending its 500 bytes, 2) the server uses a partial read.

The EOF Event

When a connection is being `read` from and receives an EOF event, it can be sure that no more data will be coming over the connection and it can stop reading. This is an important concept to understand for any IO operation.

But first, a quick bit of history: EOF stands for 'end of file'. You might say "but we're not dealing with files here...". You'd be mostly right, but need to keep in mind that *everything is a file*.

You'll sometimes see reference to the 'EOF character', but there's really no such thing. EOF is not represented as a character sequence, **EOF is more like a state event**. When a socket has no more data to write, it can `shutdown` or `close` its ability to write any more data. This results in an EOF event being sent to the reader on the other end, letting it know that no more data will be sent.

So let's bring this full circle and fix the issue we had where the client sent only 500 bytes of data while the server expected one kilobyte.

A remedy for this situation would be for the client to send their 500 bytes, then send an EOF event. The server receives this event and stops reading, even though it hasn't reached its one kilobyte limit. EOF tells it that no more data is coming.

That's the reason that this example works:

```ruby
# ./code/snippets/read_with_length.rb
require 'socket'
one_kb = 1024 # bytes

Socket.tcp_server_loop(4481) do |connection|
  # Read data in chunks of 1 kb.
  while data = connection.read(one_kb) do
    puts data
  end

  connection.close
end
```

given this client connection:

```ruby
# ./code/snippets/write_with_eof.rb
require 'socket'

client = TCPSocket.new('localhost', 4481)
client.write('gekko')
client.close
```

The simplest way for a client to send an EOF is to close its socket. If its socket is closed, it certainly won't be sending any more data!

> A quick reminder of the fact that EOF is aptly named. When you call `File#read` it behaves just like `Socket#read`. It will read data until there's no more to read. Once it's consumed the entire file, it receives an EOF event and returns the data it has.

Partial Reads

A few paragraphs back I mentioned the term 'partial read'. That's something that could have gotten us out of that last situation as well. Time to look at that.

The first method of reading data we looked at was a lazy method. When you call `read` it waits as long as possible before returning data, either until it receives its minimum length or gets an EOF. There is an alternative method of reading that takes an opposite approach. It's `readpartial`.

Calls to `readpartial`, rather than wanting to block, want to return available data immediately. When calling `readpartial` you *must* pass an integer argument, specifying the maximum length. `readpartial` will read *up to* that length. So if you tell it to read up to one kilobyte of data, but the client sends only 500 bytes, then `readpartial` will *not block*. It will return that data immediately.

Running this server:

```
# ./code/snippets/readpartial_with_length.rb
require 'socket'
one_hundred_kb = 1024 * 100

Socket.tcp_server_loop(4481) do |connection|
  begin
    # Read data in chunks of 1 hundred kb or less.
    while data = connection.readpartial(one_hundred_kb) do
      puts data
    end
  rescue EOFError
  end

  connection.close
end
```

along with this client:

```
$ tail -f /var/log/system.log | nc localhost 4481
```

will show that the server is actually streaming each bit of data as it becomes accessible, rather than waiting for one hundred kilobyte chunks. `readpartial` will happily return less than its maximum length if the data is available.

In terms of EOF, `readpartial` behaves differently than `read`. Whereas `read` simply returns when it receives EOF, `readpartial` actually raises an `EOFError` exception. Something to watch out for.

To recap, `read` is lazy, waiting as long as possible to return as much data as possible back to you. Conversely, `readpartial` is eager, returning data to you as soon as its available.

After we look at the basics of `write` we'll turn to buffers. At that point we get to answer some interesting questions like: How much should I try to read at once? Is it better to do lots of small reads or one big read?

System Calls From This Chapter

- `Socket#read` -> read(2). Behaves more like fread(3).
- `Socket#readpartial` -> read(2).

Chapter 7

Sockets Can Write

I know that at least some of you saw this coming: in order for one socket to read data another socket must write data! Rejoice!

There's really only one workhorse in terms of writing to sockets, and that's the `write` method. Its usage is pretty straightforward. Just follow your intuition.

```ruby
# ./code/snippets/write.rb
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # Simplest way to write data to a connection.
  connection.write('Welcome!')
  connection.close
end
```

There's nothing more than that to the basic `write` call. We'll get to answer some more interesting questions around writing in the next chapter when we look at buffering.

System Calls From This Chapter

- `Socket#write` -> write(2)

Chapter 8

Buffering

Here we'll answer a few key questions: How much data should I read/write with one call? If `write` returns successfully does that mean that the other end of the connection received my data? Should I split a big `write` into a bunch of smaller writes? What's the impact?

Write Buffers

Let's first talk about what really happens when you `write` data on a TCP connection.

When you call `write` and it returns, without raising an exception, this *does not* mean that the data has been successfully sent over the network and received by the client socket. When `write` returns, it acknowledges that you have left your data in the capable hands of Ruby's IO system and the underlying operating system kernel.

There is at least one layer of buffers between your application code and actual network hardware. Let's pinpoint where those are and then we'll look at how to work around them.

When `write` returns successfully, the only guarantee you have is that your data is now in the capable hands of the OS kernel. It may decide to send your data immediately, or keep it and combine it with other data for efficiency.

By default, Ruby sockets set `sync` to `true`. This skips Ruby's internal buffering [2] which would otherwise add another layer of buffers to the mix.

Why buffer at all?

All layers of IO buffering are in place for performance reasons, usually offering *big* improvements in performance.

Sending data across the network is slow [3], really slow. Buffering allows calls to `write` to return almost immediately. Then, behind the scenes, the kernel can collect all the pending writes, group them and optimize when they're sent for maximum performance to avoid flooding the network. At the network level, sending many small packets incurs a lot overhead, so the kernel batches small writes together into larger ones.

How Much to Write?

Given what we now know about buffering we can pose this question again: should I do many small `write` calls or one big `write` call?

Thanks to buffers, we don't really have to think about it. Generally you'll get better performance from writing all that you have to write and letting the kernel normalize performance by splitting things up or chunking them together. If you're doing a *really*

[2] http://jstorimer.com/2012/09/25/ruby-io-buffers.html
[3] https://gist.github.com/2841832

big `write`, think files or big data, then you'd be better off splitting up the data, lest all that stuff gets moved into RAM.

In the general case, **you'll get the best performance from writing everything you have to write in one go** and letting the kernel decide how to chunk the data. Obviously, the only way to be certain is to profile your application.

Read Buffers

It's not just writes, reads are buffered too.

When you ask Ruby to `read` data from a TCP connection and pass a maximum read length, Ruby may actually be able to receive more data than your limit allows.

In this case that 'extra' data will be stored in Ruby's internal read buffers. On the next call to `read`, Ruby will look first in its internal buffers for pending data before asking the OS kernel for more data.

How Much to Read?

The answer to this question isn't quite as straightforward as it was with write buffering, but we'll take a look at the issues and best practices.

Since TCP provides a stream of data we don't know how much is coming from the sender. This means that we'll always be making a guess when we decide on a read length.

Why not just specify a huge read length to make sure we always get all of the available data? When we specify our read length the kernel allocates some memory for us. If we specify more than we need, we end up allocating memory that we don't use. This is wasted resources.

If we specify a small read length which requires many reads to consume all of the data, we incur overhead for each system call.

So, as with most things, you'll get the best performance if you tune your program based on the data it receives. Going to receive lots of big data chunks? Then you should probably specify a bigger read length.

There's no silver bullet answer but **I've cheated a bit** and took a survey of various Ruby projects that use sockets to see what the consensus on this question is.

I've looked at Mongrel, Unicorn, Puma, Passenger, and Net::HTTP, and *all* of them do a `readpartial(1024 * 16)`. All of these web projects use 16KB as their read length.

Conversely, redis-rb uses 1KB as its read length.

You'll always get best performance through tuning your server to the data at hand but, when in doubt, 16KB seems to be a generally agreed-upon read length.

Chapter 9

Our First Client/Server

Whew.

We've now looked at establishing connections and a whole lot of information about exchanging data. Up until now we've mostly been working with very small, self-contained bits of example code. It's time to put everything we've seen together into a network server and client.

The Server

For our server we're going to write the next NoSQL solution that no one has heard of. It's just going to be a network layer on top of a Ruby hash. It's aptly named `CloudHash`.

Here's the full implementation of a simple `CloudHash` server:

```ruby
                                                    # ./code/cloud_hash/server.rb
require 'socket'

module CloudHash
  class Server
    def initialize(port)
      # Create the underlying server socket.
      @server = TCPServer.new(port)
      puts "Listening on port #{@server.local_address.ip_port}"
      @storage = {}
    end

    def start
      # The familiar accept loop.
      Socket.accept_loop(@server) do |connection|
        handle(connection)
        connection.close
      end
    end

    def handle(connection)
      # Read from the connection until EOF.
      request = connection.read

      # Write back the result of the hash operation.
      connection.write process(request)
    end

    # Supported commands:
    # SET key value
    # GET key
    def process(request)
      command, key, value = request.split
      case command.upcase
      when 'GET'
        @storage[key]

      when 'SET'
        @storage[key] = value
      end
    end
  end
end

server = CloudHash::Server.new(4481)
server.start
```

The Client

And here's a simple client:

```ruby
                                                  # ./code/cloud_hash/client.rb
require 'socket'

module CloudHash
  class Client
    class << self
      attr_accessor :host, :port
    end

    def self.get(key)
      request "GET #{key}"
    end

    def self.set(key, value)
      request "SET #{key} #{value}"
    end

    def self.request(string)
      # Create a new connection for each operation.
      @client = TCPSocket.new(host, port)
      @client.write(string)

      # Send EOF after writing the request.
      @client.close_write

      # Read until EOF to get the response.
      @client.read
    end
  end
end

CloudHash::Client.host = 'localhost'
CloudHash::Client.port = 4481
```

```
puts CloudHash::Client.set 'prez', 'obama'
puts CloudHash::Client.get 'prez'
puts CloudHash::Client.get 'vp'
```

Put It All Together

Let's stitch it all together!

Boot the server:

```
$ ruby code/cloud_hash/server.rb
```

Remember that the data structure is a Hash. Running the client will run the following operations:

```
$ tail -4 code/cloud_hash/client.rb
puts CloudHash::Client.set 'prez', 'obama'
puts CloudHash::Client.get 'prez'
puts CloudHash::Client.get 'vp'

$ ruby code/cloud_hash/client.rb
```

Thoughts

So what have we done here? We've wrapped a Ruby hash with a network API, but not even the whole `Hash` API, just the getter/setter. A good chunk of the code is boilerplate networking stuff, so it should be easy to see how you could extend this example to expose more of the `Hash` API.

I commented the code so you can get an idea of why things are done the way they are, I made sure to stick to concepts that we've already seen, such as establishing connections, EOF, etc.

But overall, `CloudHash` is kind of a kludge. In the last few chapters we've gone over the basics of establishing connections and exchanging data. Both of those things were applied here. What's missing from this example is best practices about architecture patterns, design, and some advanced features we haven't seen yet.

For example, notice that the client has to initiate a new connection for each request it sends? If you wanted to send a bunch of requests all in a row,, each would require its own connection. The current server design requires this. It processes one command from the client socket and then closes it.

There's no reason why this needs to be the case. Establishing connections incurs overhead and it's certainly possible for `CloudHash` to handle multiple requests on the same connection.

This can be remedied in a few different ways. The client/server could communicate using a simple message protocol that doesn't require sending EOF to delimit messages. This change would allow for multiple requests on a single connection but the server will still process each client connection in sequence. If one client is sending a lot of requests or leaving their connection open for a long time, then other clients will not be able to interact with the server.

We can resolve this by building some form of concurrency into the server. The rest of the book builds on the basic knowledge you have thus far and focuses on helping you write efficient, understandable, and sane network programs. `CloudHash`, as it stands, *does not* provide a good example of how socket programming should be done.

Chapter 10

Socket Options

To kick off this set of chapters on advanced techniques we'll look at socket options. Socket options are a low-level way to configure system-specific behaviour on sockets. So low-level, in fact, that Ruby doesn't provide a fancy wrapper around the system calls.

SO_TYPE

Let's begin by having a look at retrieving a socket option: the socket type.

```ruby
# ./code/snippets/getsockopt.rb
require 'socket'

socket = TCPSocket.new('google.com', 80)
# Get an instance of Socket::Option representing the type of the socket.
opt = socket.getsockopt(Socket::SOL_SOCKET, Socket::SO_TYPE)

# Compare the integer representation of the option to the integer
# stored in Socket::SOCK_STREAM for comparison.
opt.int == Socket::SOCK_STREAM #=> true
opt.int == Socket::SOCK_DGRAM #=> false
```

A call to `getsockopt` returns an instance of `Socket::Option`. When working at this level everything resolves to integers. So `SocketOption#int` gets the underlying integer associated with the return value.

In this case I'm retrieving the socket type (remember specifying this back when creating our first socket?), so I compare the `int` value against the various `Socket` type constants and find that it's a `STREAM` socket.

Remember that Ruby *always* offers memoized symbols in place of these constants. The above can also be written as:

```ruby
# ./code/snippets/getsockopt_wrapper.rb
require 'socket'

socket = TCPSocket.new('google.com', 80)
# Use the symbol names rather than constants.
opt = socket.getsockopt(:SOCKET, :TYPE)
```

SO_REUSE_ADDR

This is a common option that *every* server should set.

The `SO_REUSE_ADDR` option tells the kernel that it's OK for another socket to `bind` to the same local address that the server is using *if it's currently in the TCP* `TIME_WAIT` *state*.

TIME_WAIT state?

This can come about when you `close` a socket that has pending data in its buffers. Remember calling `write` only guarantees that your data has entered the buffer layers? When you `close` a socket its pending data is *not* discarded.

Behind the scenes, the kernel leaves the connection open long enough to send out that pending data. This means that it actually has to send the data, and then wait for acknowledgement of receipt from the receiving end in case some data needs retransmitting.

If you close a server with pending data and immediately try to `bind` another socket on the same address (such as if you reboot the server immediately) then an `Errno::EADDRINUSE` will be raised unless the pending data has been accounted for. Setting `SO_REUSE_ADDR` will circumvent this problem and allow you to `bind` to an address still in use by another socket in the `TIME_WAIT` state.

Here's how to switch it on:

```ruby
# ./code/snippets/reuseaddr.rb
require 'socket'

server = TCPServer.new('localhost', 4481)
server.setsockopt(:SOCKET, :REUSEADDR, true)

server.getsockopt(:SOCKET, :REUSEADDR) #=> true
```

Note that `TCPServer.new`, `Socket.tcp_server_loop` and friends enable this option by default.

For a complete listing of socket options avabilable on your system look at setsockopt(2).

System Calls From This chapter

- `Socket#setsockopt` -> setsockopt(2)
- `Socket#getsockopt` -> getsockopt(2) :

Chapter 11

Non-blocking IO

This chapter is about non-blocking IO. Note: this is different from asynchronous or evented IO. If you don't know the difference, it should become clear as you progress through the rest of the book.

Non-blocking IO goes hand-in-hand with the next chapter on *Multiplexing Connections*, but we'll look at this first in isolation because it can be useful on its own.

Non-blocking Reads

Do you remember a few chapters back when we looked at `read`? I noted that `read` blocked until it received EOF or was able to receive a minimum number of bytes. This may result in a lot of blocking when a client doesn't send EOF. This blocking behaviour can be partly circumvented by `readpartial`, which returns any available data immediately. But `readpartial` will still block if there's no data available. For a read operation that will *never* block you want `read_nonblock`.

Much like `readpartial`, `read_nonblock` requires an Integer argument specifying the maximum number of bytes to read. Remember that `read_nonblock`, like `readpartial`, might return less than the maximum amount of bytes if that's what's available. It works something like this:

```ruby
# ./code/snippets/read_nonblock.rb
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  loop do
    begin
      puts connection.read_nonblock(4096)
    rescue Errno::EAGAIN
      retry
    rescue EOFError
      break
    end
  end

  connection.close
end
```

Boot up the same client we used previously that never closes its connection:

```
$ tail -f /var/log/system.log | nc localhost 4481
```

Even when there's no data being sent to the server the call to `read_nonblock` is still returning immediately. In fact, it's raising an `Errno::EAGAIN` exception. Here's what my manpages have to say about `EAGAIN`:

```
The file was marked for non-blocking I/O, and no data were ready to be
read.
```

Makes sense. This differs from `readpartial` which would have just blocked in that situation.

So what should you do when you get this error and your socket would otherwise block? In this example we entered a busy loop and continued to `retry` over and over again. This was just for demonstration purposes and isn't the proper way to do things.

The proper way to retry a blocked read is using `IO.select`:

```
begin
  connection.read_nonblock(4096)
rescue Errno::EAGAIN
  IO.select([connection])
  retry
end
```

This achieves the same effect as spamming `read_nonblock` with `retry`, but with less wasted cycles. Calling `IO.select` with an Array of sockets as the first argument will block until one of the sockets becomes readable. So, `retry` will only be called when the socket has data available for reading. We'll cover `IO.select` in more detail in the next chapter.

In this example we've re-implemented a blocking `read` method using non-blocking methods. This, in itself, isn't useful. But using `IO.select` gives the flexibility to monitor multiple sockets simultaneously or periodically check for readability while doing other work.

When would a read block?

The read_nonblock method first checks Ruby's internal buffers for any pending data. If there's some there it's returned immediately.

It then asks the kernel if there's any data available for reading using select(2). If the kernel says that there's some data available, whether it be in the kernel buffers or over the network, that data is then consumed and returned. Any other condition would cause a read(2) to block and, thus, raise an exception from read_nonblock.

Non-blocking Writes

Non-blocking writes have some very important differences from the write call we saw earlier. The most notable is that it's possible for write_nonblock to return a partial write, whereas write will always take care of writing all of the data that you send it.

Let's boot up a throwaway server using netcat to show this behaviour:

```
$ nc -l localhost 4481
```

Then we'll boot up this client that makes use of write_nonblock:

```ruby
# ./code/snippets/write_nonblock.rb
require 'socket'

client = TCPSocket.new('localhost', 4481)
payload = 'Lorem ipsum' * 10_000

written = client.write_nonblock(payload)
written < payload.size #=> true
```

When I run those two programs against each other, I routinely see `true` being printed out from the client side. In other words it's returning an Integer that's less than the full size of the payload data. The `write_nonblock` method returned because it entered some situation where it would block, so it didn't write any more data and returned an Integer, letting us know how much was written. It's now our responsibility to write the rest of the data that remains unsent.

The behaviour of `write_nonblock` is the same as the write(2) system call. It writes as much data as it can and returns the number of bytes written. This differs from Ruby's `write` method which may call write(2) several times to write all of the data requested.

So what should you do when one call couldn't write all of the requested data? Try again to write the missing portion, obviously. But don't do it right away. If the underlying write(2) would still block then you'll get an `Errno::EAGAIN` exception raised. The answer lies again with `IO.select`, it can tell us when a socket is writable, meaning it can write without blocking.

```ruby
# ./code/snippets/retry_partial_write.rb
require 'socket'

client = TCPSocket.new('localhost', 4481)
payload = 'Lorem ipsum' * 10_000

begin
  loop do
    bytes = client.write_nonblock(payload)

    break if bytes >= payload.size
    payload.slice!(0, bytes)
    IO.select(nil, [client])
  end

rescue Errno::EAGAIN
  IO.select(nil, [client])
  retry
end
```

Here we make use of the fact that calling `IO.select` with an Array of sockets as the second argument will block until one of the sockets becomes writable.

The loop in the example deals properly with partial writes. When `write_nonblock` returns an Integer less than the size of the payload we slice that data from the payload and go back around the loop when the socket becomes writable again.

When would a write block?

The underlying write(2) can block in two situations:

1. The receiving end of the TCP connection has not yet acknowledged receipt of pending data, and we've sent as much data as is allowed. Due to the algorithms TCP uses for congestion control, it ensures that the network is never flooded with packets. If the data is taking a long time to reach the receiving end of the TCP connection, then care is taken not to flood the network with more data than can be handled.
2. The receiving end of the TCP connection cannot yet handle more data. Even once the other end acknowledges receipt of the data it still must clear its data 'window' in order that it may be refilled with more data. This refers to the kernel's read buffers. If the receiving end is not processing the data it's receiving then the congestion control algorithms will force the sending end to block until the client is ready for more data.

Non-blocking Accept

There are non-blocking variants of other methods, too, besides `read` and `write`, though they're the most commonly used.

An `accept_nonblock` is very similar to a regular `accept`. Remember how I said that `accept` just pops a connection off of the listen queue? Well if there's nothing on that queue then `accept` would block. In this situation `accept_nonblock` would raise an `Errno::EAGAIN` rather than blocking.

Here's an example:

```
# ./code/snippets/accept_nonblock.rb
require 'socket'

server = TCPServer.new(4481)

loop do
  begin
    connection = server.accept_nonblock
  rescue Errno::EAGAIN
    # do other important work
    retry
  end
end
```

Non-blocking Connect

Think you can guess what the `connect_nonblock` method does by now? Then you're in for a surprise! `connect_nonblock` behaves a bit differently than the other non-blocking IO methods.

Whereas the other methods *either* complete their operation or raise an appropriate exception, `connect_nonblock` leaves its operation in progress *and* raises an exception.

If `connect_nonblock` cannot make an immediate connection to the remote host, then it actually lets the operation continue in the background and raises `Errno::EINPROGRESS` to notify us that the operation is still in progress. In the next chapter we'll see how we can be notified when this background operation completes. For now, a quick example:

```
                                                    # ./code/snippets/connect_nonblock.rb
require 'socket'

socket = Socket.new(:INET, :STREAM)
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')

begin
  # Initiate a nonblocking connection to google.com on port 80.
  socket.connect_nonblock(remote_addr)
rescue Errno::EINPROGRESS
  # Operation is in progress.
rescue Errno::EALREADY
  # A previous nonblocking connect is already in progress.
rescue Errno::ECONNREFUSED
  # The remote host refused our connect.
end
```

Chapter 12

Multiplexing Connections

Connection multiplexing refers to working with multiple active sockets at the same time. This doesn't specifically refer to doing any work in parallel and is not related to multi-threading. An example will make it clear.

Given the techniques seen so far, let's imagine how we might write a server that needs to process available data on several TCP connections at any given time. We'd probably use our newfound knowledge of non-blocking IO to keep ourselves from getting stuck blocking on any particular socket.

```
                                                  # ./code/snippets/naive_multiplexing.rb
# Given an Array of connections.
connections = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

# We enter an endless loop.
loop do
  # For each connection...
  connections.each do |conn|
    begin
      # Attempt to read from each connection in a non-blocking way,
      # processing any data received, otherwise trying the next
      # connection.
      data = conn.read_nonblock(4096)
      process(data)
    rescue Errno::EAGAIN
    end
  end
end
```

Does this work? It does! But it's a very busy loop.

Each call to `read_nonblock` uses at least one system call and the server will be wasting a lot of cycles trying to read data when there is none. Remember that I said `read_nonblock` checks if there's any data available using select(2)? Well, there's a Ruby wrapper so that we can use select(2) directly for our own purposes.

select(2)

Here's the saner method of processing available data on multiple TCP connections:

```ruby
# ./code/snippets/sane_multiplexing.rb
# Given an Array of connections.
connections = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

loop do
  # Ask select(2) which connections are ready for reading.
  ready = IO.select(connections)

  # Read data only from the available connections.
  readable_connections = ready[0]
  readable_connections.each do |conn|
    data = conn.readpartial(4096)
    process(data)
  end
end
```

This example uses `IO.select` to greatly reduce the overhead of handling multiple connections. The whole purpose of `IO.select` is take in some `IO` objects and tell you which of those are ready to be read from or written to so you don't have to take shots in the dark like we did above.

Let's review some properties of `IO.select`.

It tells you when file descriptors are ready for reading or writing. In the above example we only passed one argument to `IO.select`, but there are actually three important Arrays that `IO.select` takes as arguments.

```ruby
# ./code/snippets/select_args.rb
for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

IO.select(for_reading, for_writing, for_writing)
```

The first argument is an Array of `IO` objects which you want to read from. The second argument is an Array of `IO` objects which you want to write to. The third argument is an Array of `IO` objects for which you are interested in exceptional conditions. The vast majority of applications can ignore the third argument unless you're interested in out-of-band data (more on that in the *Urgent Data* chapter). Note that even if you're interested in reading from a single `IO` object you still must put it in an Array to pass to `IO.select`.

It returns an Array of Arrays. `IO.select` returns a nested array with three elements that correspond to its argument list. The first element will contain `IO` objects that can be read from without blocking. Note that this will be a subset of the Array of `IO` objects passed in as the first argument. The second element will contain `IO` objects that can be written to without blocking, and the third element will contain `IO` objects which have applicable exceptional conditions.

```
# ./code/snippets/select_returns.rb
for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

ready = IO.select(for_reading, for_writing, for_writing)

# One Array is returned for each Array passed in as an argument.
# In this case none of the connections in for_writing were writable
# and one of connections in for_reading was readable.
puts ready #=> [[<TCPSocket>], [], []]
```

It's blocking. `IO.select` is a synchronous method call. Using it like we've seen thus far will cause it to block until the status of one of the passed-in `IO` objects changes. At this point it will return immediately. If multiple statuses have changed then all will be returned via the nested Array.

But IO.select will also take a fourth argument, a timeout value in seconds. This will prevent IO.select from blocking indefinitely. Pass in an Integer or Float value to specify a timeout. If the timeout is reached before any of the IO statuses have changed, IO.select will return nil.

```
# ./code/snippets/select_timeout.rb
for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

timeout = 10
ready = IO.select(for_reading, for_writing, for_writing, timeout)

# In this case IO.select didn't detect any status changes in 10 seconds,
# thus returned nil instead of a nested Array.
puts ready #=> nil
```

You can also pass plain Ruby objects to IO.select, so long as they respond to to_io and return an IO object. This is useful so that you don't need to maintain a mapping of IO object -> your domain object. IO.select can work with your plain Ruby objects if they implement this to_io method.

Events Other Than Read/Write

So far we've just looked at monitoring readable and writable state with IO.select, but it can actually be shoehorned into a few other places.

EOF

If you're monitoring a socket for readability and it receives an EOF, it will be returned as part of the readable sockets Array. Depending on which variant of read(2) you use at that point you might get an `EOFError` or `nil` when trying to read from it.

Accept

If you're monitoring a server socket for readability and it receives an incoming connection, it will be returned as part of the readable sockets Array. Obviously, you'll need to have logic to handle these kinds of sockets specially and use `accept` rather than `read`.

Connect

This one is probably the most interesting of the bunch. In the last chapter we looked at `connect_nonblock` and noted that it raised `Errno::EINPROGRESS` if it couldn't connect immediately. Using `IO.select` we can figure out if that background connect has yet completed:

```ruby
# ./code/snippets/multiplexing_connect.rb
require 'socket'

socket = Socket.new(:INET, :STREAM)
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')

begin
  # Initiate a nonblocking connection to google.com on port 80.
  socket.connect_nonblock(remote_addr)
rescue Errno::EINPROGRESS
  IO.select(nil, [socket])

  begin
    socket.connect_nonblock(remote_addr)
  rescue Errno::EISCONN
    # Success!
  rescue Errno::ECONNREFUSED
    # Refused by remote host.
end
```

The first part of this snippet is the same as last chapter. Try to do a `connect_nonblock` and rescue `Errno::EINPROGRESS`, which signifies that the connect is happening in the background. Then we enter the new code.

We ask `IO.select` to monitor the socket for changes to its writable status. When that changes, we know that the underlying connect is complete. In order to figure out the status, we just try `connect_nonblock` again! If it raises `Errno::EISCONN` this tells us that the socket is already connected to the remote host. Success! A different exception signifies an error condition in connecting to the remote host.

This fancy bit of code actually emulates a *blocking* `connect`. Why? Partly to show you what's possible, but you can also imagine sticking your own code into this process. You could initiate the `connect_nonblock`, go off and do some other work, then call `IO.select` with a timeout. If the underlying connect isn't finished then you can continue doing other work and check `IO.select` again later.

We can actually use this little technique to build a pretty simple port scanner [2] in Ruby. A port scanner attempts to make connections to a range of ports on a remote host and tells you which ones were open to connections.

[2] http://en.wikipedia.org/wiki/Port_scanner

```
                                                          # ./code/port_scanner.rb
require 'socket'

# Set up the parameters.
PORT_RANGE = 1..512
HOST = 'archive.org'
TIME_TO_WAIT = 5 # seconds

# Create a socket for each port and initiate the nonblocking
# connect.
sockets = PORT_RANGE.map do |port|
  socket = Socket.new(:INET, :STREAM)
  remote_addr = Socket.pack_sockaddr_in(port, HOST)

  begin
    socket.connect_nonblock(remote_addr)
  rescue Errno::EINPROGRESS
  end

  socket
end

# Set the expiration.
expiration = Time.now + TIME_TO_WAIT

loop do
  # We call IO.select and adjust the timeout each time so that we'll never
  # be waiting past the expiration.
  _, writable, _ = IO.select(nil, sockets, nil, expiration - Time.now)
  break unless writable

  writable.each do |socket|
    begin
      socket.connect_nonblock(socket.remote_address)
    rescue Errno::EISCONN
      # If the socket is already connected then we can count this as a success.
      puts "#{HOST}:#{socket.remote_address.ip_port} accepts connections..."
      # Remove the socket from the list so it doesn't continue to be
      # selected as writable.
      sockets.delete(socket)
    rescue Errno::EINVAL
      sockets.delete(socket)
    end
  end
end
```

This bit of code takes advantage of `connect_nonblock` by initiating several hundred connections at once. It then monitors all of these using `IO.select` and ultimately verifies which we were able to connect to successfully. Here's the output I got when running this against archive.org:

```
archive.org:25 accepts connections...
archive.org:22 accepts connections...
archive.org:80 accepts connections...
archive.org:443 accepts connections...
```

Notice that the results aren't necessarily in order. The first connections that finish the process are printed first. This a pretty common group of open ports, port 25 is reserved for SMTP, port 22 for SSH, port 80 for HTTP and port 443 for HTTPS.

High Performance Multiplexing

`IO.select` ships with Ruby's core library. But it's the only solution for multiplexing that ships with Ruby. Most modern OS kernels support multiple methods of multiplexing. Almost invariably, select(2) is the oldest and least performing of these methods.

`IO.select` will perform well with few connections, but its performance is linear to the number of connections it monitors. As it monitors more connections its performance will continue to degrade. Moreover, the select(2) system call is limited by something called `FD_SETSIZE`, a C macro that's defined as part of your local C library. select(2) is unable to monitor a file descriptor number higher than the number in `FD_SETSIZE` (1024 on most systems). So `IO.select` will be limited to monitoring at most 1024 `IO` objects.

There are, of course, alternatives.

The poll(2) system call provides some slight differences over select(2) but is more or less on par. The (Linux) epoll(2) and (BSD) kqueue(2) system calls provide a more performing, modern alternative to select(2) and poll(2). For example, a high-performance networking toolkit like EventMachine will favour epoll(2) or kqueue(2) where possible.

Rather than trying to give examples of these particular system calls I'll point you to the `nio4r` Ruby gem [3], which provides a common interface to the all of these multiplexing solutions, favouring the most performant one available on your system.

[3] https://github.com/tarcieri/nio4r

Chapter 13

Nagle's algorithm

Nagle's algorithm is a so-called optimization applied to all TCP connections by default.

This optimization is most applicable to applications which don't do buffering and send very small amounts of data at a time. As such, it's often disabled by servers where those criteria don't apply. Let's review the algorithm:

After a program writes to a socket there are three possible outcomes:

1. If there's sufficient data in the local buffers to comprise an entire TCP packet then send it all immediately.
2. If there's no pending data in the local buffers and no pending acknowledgement of receipt from the receiving end, then send it immediately.
3. If there's a pending acknowledgement of receipt from the other end and not enough data to comprise an entire TCP packet, then put the data into the local buffer.

This algorithm guards against sending many tiny TCP packets. It was originally designed to combat protocols like telnet where one key stroke is entered at a time and otherwise each character could be sent across the network without delay.

If you're working with a protocol like HTTP where the request/response are usually sufficiently large enough to comprise at least one TCP packet, this algorithm will typically have have no effect except to slow down the last packet sent. The algorithm is meant to guard against shooting yourself in the foot during very specific situations, such as implementing telnet. Given Ruby's buffering and the most common kinds of protocols implemented on TCP, you probably want to disable this algorithm.

For example, every Ruby web server disables this option. Here's how it can be done:

```ruby
# ./code/snippets/disable_nagle.rb
require 'socket'

server = TCPServer.new(4481)

# Disable Nagle's algorithm. Tell the server to send with 'no delay'.
server.setsockopt(Socket::IPPROTO_TCP, Socket::TCP_NODELAY, 1)
```

Chapter 14

Framing Messages

One thing that we haven't talked about yet is how to format the messages that are exchanged between server and client.

One problem we had with `CloudHash` was that the client had to open a new connection for every command it wanted to send to the server. The main reason for this is that the client/server had no agreed-upon way to frame the beginning and end of messages, so they had to fall back to using EOF to signify the end of a message.

While this technically 'got the job done', it wasn't ideal. Opening a new connection for each command adds unnecessary overhead. It's certainly possible to send multiple messages over the same TCP connection, but if you're going to leave the connection open, you need some way of signaling that one message is ending and another is beginning.

> This idea of reusing connections across multiple messages is the same concept behind the familiar keep-alive feature of HTTP. By leaving the connection open for multiple requests (and having an agreed-upon method of framing messages) resources can be saved by not opening new connections.

There are, literally, an infinite number of choices for framing your messages. Some are very complicated; some are simple. It all depends on how you want to format your messages.

Protocol vs. Message

I keep talking about messages, which I see as distinct from protocols. For example, the HTTP protocol defines both the message boundaries (a series of newlines) as well as a protocol for the content of the message involving a request line, headers, etc.

A protocol defines how your messages should be formatted, whereas this chapter is concerned with how to separate your messages from one another on the TCP stream.

Using newlines

Using newlines is a really simple way to frame your messages. If you know, for certain, that your application client and server will be running on the same operating system, you can even fall back to using `IO#gets` and `IO#puts` on your sockets to send messages with newlines.

Let's rewrite the relevant part of the `CloudHash` server to frame messages with newlines instead of EOFs:

```
def handle(connection)
  loop do
    request = connection.gets
    break if request == 'exit'
    connection.puts process(request)
  end
end
```

The relevant changes to the server are just the addition of the `loop` and the change from `read` to `gets`. In this way the server will process as many requests as the client wishes to send until it sends the 'exit' request.

A more robust approach would be to use `IO.select` and wait for events on the connection. Currently the server would come crashing down if the client socket disconnected without first sending the 'exit' request.

The client would then send its requests with something like:

```
  def initialize(host, port)
    @connection = TCPSocket.new(host, port)
  end

  def get
    request "GET #{key}"
  end

  def set
    request "SET #{key} #{value}"
  end

  def request(string)
    @connection.puts(string)

    # Read until receiving a newline to get the response.
    @connection.gets
  end
```

Note that the client no longer uses class methods. Now that our connection can persist across requests, we can encapsulate a single connection in an instance of an object and just call methods on that object.

Newlines and Operating Systems

Remember that I said it was permissible to use `gets` and `puts` if you're certain that the client/server will run on the same operating system? Let me explain why.

If you look at the documentation for `gets` and `puts` it says that it uses `$/` as the default line delimiter. This variable holds the value `\n` on Unix systems but holds the value of `\r\n` on Windows systems. Hence my warning, one system using `puts` may not be compatible with another using `gets`. If you use this method then ensure that you pass an argument to those methods with an explicit line delimiter so that they're compatible.

One real-world protocol that uses newlines to frame messages is HTTP. Here's an example of a short HTTP request:

```
GET /index.html HTTP/1.1\r\n
Host: www.example.com\r\n
\r\n
```

In this example the newlines are made explicit with the escape sequence `\r\n`. This sequence of newlines must be respected by any HTTP client/server, regardless of operating system.

This method certainly works, but it's not the only way.

Using A Content Length

Another method of framing messages is to specify a content length.

With this method the sender of the message first denotes the size of their message, packs that into a fixed-width integer representation and sends that over the connection, immediately followed by the message itself. The receiver of the message will read the fixed-width integer to begin with. This gets them the message size. Then the receiver can read the exact number of bytes specified in the message size to get the whole message.

Here's how we might change the relevant part of the `CloudHash` server to use this method:

```ruby
# This gets us the size of a random fixed-width integer.
SIZE_OF_INT = [11].pack('i').size

def handle(connection)
  # The message size is packed into a fixed-width. We
  # read it and unpack it.
  packed_msg_length = connection.read(SIZE_OF_INT)
  msg_length = packed_msg_length.unpack('i').first

  # Fetch the full message given its length.
  request = connection.read(msg_length)
  connection.write process(request)
end
```

The client would send a request using something like:

```
payload = 'SET prez obama'

# Pack the message length into a fixed-width integer.
msg_length = payload.size
packed_msg_length = [msg_length].pack('i')

# Write the length of the message, followed immediately
# by the message itself.
connection.write(packed_msg_length)
connection.write(payload)
```

The client packs the message length as a native endian integer. This is important because it guarantees that any given integer will be packed into the same number of bytes. Without this guarantee the server wouldn't know whether to read a 2, 3, 4, or even higher digit number to represent the message size. Using this method the client/server always communicate using the same number of bytes for the message size.

As you can see it's a bit more code, but this method doesn't use any wrapper methods like `gets` or `puts`, just the basic IO operations like `read` and `write`.

Chapter 15

Timeouts

Timeouts are all about tolerance. How long are you willing to wait for your socket to connect? To read? To write?

All of these answers are a function of your tolerance. High performance network programs typically aren't willing to wait for operations that aren't going to finish. It's assumed that if your socket can't write its data in the first 5 seconds then there's a problem and some other behaviour should take over.

Unusable Options

If you've spent any time reading Ruby code you've probably seen the `timeout` library that comes with the standard library. Although that library tends to get used with socket programming in Ruby, I'm *not* even going to talk about it here because there are better ways! The `timeout` library provides a general purpose timeout mechanism, but your operating system comes with socket-specific timeout mechanisms that are more performing and more intuitive.

Your operating system also offers native socket timeouts that can be set via the `SNDTIMEO` and `RCVTIMEO` socket options. *But*, as of Ruby 1.9, this feature is no longer functional. Due to the way that Ruby handles blocking IO in the presence of threads, it wraps all socket operations around a poll(2), which mitigates the effect of the native socket timeouts. So those are unusable too.

What's left?

IO.select

Ah, let's call `IO.select` old faithful, eh? So many uses.

We've already seen how to use `IO.select` in previous chapters. Here's how you can use it for timeouts.

```ruby
# ./code/snippets/read_timeout.rb
require 'socket'
require 'timeout'

timeout = 5 # seconds

Socket.tcp_server_loop(4481) do |connection|

  begin
    # Initiate the initial read(2). This is important because
    # it requires data be requested on the socket and circumvents
    # a select(2) call when there's already data available to read.
    connection.read_nonblock(4096)

  rescue Errno::EAGAIN
    # Monitor the connection to see if it becomes readable.
    if IO.select([connection], nil, nil, timeout)
      # IO.select will actually return our socket, but we
      # don't care about the return value. The fact that
      # it didn't return nil means that our socket is readable.
      retry

    else
      raise Timeout::Error
    end
  end

  connection.close
end
```

I actually required `timeout` in this case just to get access to that handy `Timeout::Error` constant.

Accept Timeout

As we've seen before `accept` works very nicely with `IO.select`. If you need to do a timeout around `accept` it would look just like it did for `read`.

```
server = TCPServer.new(4481)
timeout = 5 # seconds

begin
  server.accept_nonblock
rescue Errno::EAGAIN
  if IO.select([server], nil, nil, timeout)
    retry
  else
    raise Timeout::Error
  end
end
```

Connect Timeout

In this case, doing a timeout around a connect works much like the other examples we've seen.

```
                                        # ./code/snippets/connect_timeout.rb
require 'socket'
require 'timeout'

socket = Socket.new(:INET, :STREAM)
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')
timeout = 5 # seconds

begin
  # Initiate a nonblocking connection to google.com on port 80.
  socket.connect_nonblock(remote_addr)

rescue Errno::EINPROGRESS
  # Indicates that the connect is in progress. We monitor the
  # socket for it to become writable, signaling that the connect
  # is completed.
  #
  # Once it retries the above block of code it
  # should fall through to the EISCONN rescue block and end up
  # outside this entire begin block where the socket can be used.
  if IO.select(nil, [socket], nil, timeout)
    retry
  else
    raise Timeout::Error
  end

rescue Errno::EISCONN
  # Indicates that the connect is completed successfully.
end

socket.write("ohai")
socket.close
```

These `IO.select` based timeout mechanisms are commonly used, even in Ruby's standard library, and offer more stability than something like native socket timeouts.

Chapter 16

DNS Lookups

Timeouts are great to keep for your own code under control, but there are factors that you have less control of.

Take this example client connection:

```ruby
# ./code/snippets/client_easy_way.rb
require 'socket'

socket = TCPSocket.new('google.com', 80)
```

We know that inside that constructor Ruby makes a call to `connect`. Since we're passing a hostname, rather than IP address, Ruby needs to do a DNS lookup to resolve that hostname to a unique address it can connect to.

The kicker? A slow DNS server can block your entire Ruby process. This is a bummer for multi-threaded environments.

MRI and the GIL

The standard Ruby implementation (MRI) has something called a global interpreter lock (GIL). The GIL is there for safety. It ensures that the Ruby interpreter is only ever doing one potentially dangerous thing at a time. This really comes into play in a multi-threaded environment. While one thread is active *all other threads are blocked*. This allows MRI to be written with safer, simpler code.

Thankfully, the GIL *does* understand blocking IO. If you have a thread that's doing blocking IO (eg. a blocking `read`), MRI will release the GIL and let another thread continue execution. When the blocking IO call is finished, the thread lines up for another turn to execute.

MRI is a little less forgiving when it comes to C extensions. Any time that a library uses the C extension API, the GIL blocks any other code from execution. There is no exception for blocking IO here, if a C extension is blocking on IO then all other threads will be blocked.

The key to the issue at hand here is that, out of the box, Ruby uses a C extension for DNS lookups. Hence, if that DNS lookup is blocking for a long time MRI *will not* release the GIL.

resolv

Thankfully, Ruby provides a solution to this in the standard library. The 'resolv' library provides a pure-Ruby replacement for DNS lookups. This allows MRI to release the GIL for long-blocking DNS lookups. This is a big win for multi-threaded environments.

The 'resolv' library has its own API, but the standard library also provides a library that monkeypatches the `Socket` classes to use 'resolv'.

```
require 'resolv' # the library
require 'resolv-replace' # the monkey patch
```

I recommend this whenever you're doing socket programming in a multi-threaded environment.

Chapter 17

SSL Sockets

SSL provides a mechanism for exchanging data securely over sockets using public key cryptography.

SSL sockets don't replace TCP sockets, but they allow you to 'upgrade' plain ol' insecure socket to secure SSL sockets. You can add a secure layer, if you will, on top of your TCP sockets.

Note that a socket can be upgraded to SSL, but a single socket can't do both SSL and non-SSL communication. When using SSL, end-to-end communication with the receiver will all be done using SSL. Otherwise there's no security.

For services that need to be available over SSL, as well as insecure over plain TCP, two ports (and two sockets) are required. HTTP is a common example of this: insecure HTTP traffic happens on port 80 by default, whereas HTTPS (HTTP over SSL) communication happens on port 443 by default.

So any TCP socket can be transformed into an SSL socket. In Ruby this is most often done using the `openssl` library included in the standard library. Here's an example:

```ruby
# ./code/snippets/ssl_server.rb
require 'socket'
require 'openssl'

def main
  # Build the TCP server.
  server = TCPServer.new(4481)

  # Build the SSL context.
  ctx = OpenSSL::SSL::SSLContext.new
  ctx.cert, ctx.key = create_self_signed_cert(
    1024,
    [['CN', 'localhost']],
    "Generated by Ruby/OpenSSL"
  )
  ctx.verify_mode = OpenSSL::SSL::VERIFY_PEER

  # Build the SSL wrapper around the TCP server.
  ssl_server = OpenSSL::SSL::SSLServer.new(server, ctx)

  # Accept connections on the SSL socket.
  connection = ssl_server.accept

  # Treat it like any other connection.
  connection.write("Bah now")
  connection.close
end

# This code is unabashedly taken straight from webrick/ssl.
# It generates a self-signed SSL certificate suitable for use
# with a Context object.
def create_self_signed_cert(bits, cn, comment)
  rsa = OpenSSL::PKey::RSA.new(bits){|p, n|
    case p
    when 0; $stderr.putc "."  # BN_generate_prime
    when 1; $stderr.putc "+"  # BN_generate_prime
    when 2; $stderr.putc "*"  # searching good prime,
      # n = #of try,
      # but also data from BN_generate_prime
    when 3; $stderr.putc "\n" # found good prime, n==0 - p, n==1 - q,
      # but also data from BN_generate_prime
    else;   $stderr.putc "*"  # BN_generate_prime
    end
  }
  cert = OpenSSL::X509::Certificate.new
  cert.version = 2
  cert.serial = 1
  name = OpenSSL::X509::Name.new(cn)
  cert.subject = name
  cert.issuer = name
  cert.not_before = Time.now
  cert.not_after = Time.now + (365*24*60*60)
  cert.public_key = rsa.public_key

  ef = OpenSSL::X509::ExtensionFactory.new(nil,cert)
  ef.issuer_certificate = cert
  cert.extensions = [
    ef.create_extension("basicConstraints","CA:FALSE"),
    ef.create_extension("keyUsage", "keyEncipherment"),
    ef.create_extension("subjectKeyIdentifier", "hash"),
    ef.create_extension("extendedKeyUsage", "serverAuth"),
    ef.create_extension("nsComment", comment),
  ]
  aki = ef.create_extension("authorityKeyIdentifier",
                            "keyid:always,issuer:always")
  cert.add_extension(aki)
  cert.sign(rsa, OpenSSL::Digest::SHA1.new)
  return [ cert, rsa ]
end

main
```

That bit of code generates a self-signed SSL certificate and uses that to support the SSL connection. The certificate is the cornerstone of security for SSL. Without it you're basically just using a fancy insecure connection.

Likewise, setting `verify_mode = OpenSSL::SSL::VERIFY_PEER` is essential to the security of your connection. Many Ruby libraries default that value to `OpenSSL::SSL::VERIFY_NONE`. This is a more lenient setting that will allow non-verified SSL certificates, forgoing much of the assumed security provided by the certificates. This issue has been discussed [2] at length in the Ruby community.

So once you have that server running you can attempt to connect to it using a regular ol' TCP connection with netcat:

```
$ echo hello | nc localhost 4481
```

Upon doing so your server will crash with an `OpenSSL::SSL::SSLError`. This is a good thing!

The server refused to accept a connection from an insecure client and, so, raised an exception. Boot up the server again and we'll connect to it using an SSL-secured Ruby client:

[2] http://www.rubyinside.com/how-to-cure-nethttps-risky-default-https-behavior-4010.html

```ruby
# ./code/snippets/ssl_client.rb
require 'socket'
require 'openssl'

# Create the TCP client socket.
socket = TCPSocket.new('0.0.0.0', 4481)

ssl_socket = OpenSSL::SSL::SSLSocket.new(socket)
ssl_socket.connect

ssl_socket.read
```

Now you should see both programs exit successfully. In this case there was successful SSL communication between server and client.

In a typical production setting you wouldn't generate a self-signed certificate (that's suitable only for development/testing). You'd usually buy a certificate from a trusted source . The SSL source would provide you with the `cert` and `key` that you need for secure communication and those would take the place of the self-signed certificate in our example.

Chapter 18

Network Architecture Patterns

Where the previous chapters covered the basics and the 'need-to-knows', this set of chapters turns to best practices and real-world examples. I consider the book up until this point to be like reference documentation: you can refer back to it and remind yourself how to use certain features or solve certain problems, but only if you know what you're looking for.

If you were tasked with building an FTP server in Ruby, just knowing the first half of this book would certainly help, but it wouldn't lead you to creating great software.

Though you know the building blocks you haven't yet seen common ways to structure networked applications. How should concurrency be handled? How should errors be handled? What's the best way to handle slow clients? How can you make most efficient use of resources?

These are the kinds of questions that this section of the book aims to answer. We'll begin by looking at 6 network architectures patterns and apply these to an example project.

The Muse

Rather than just use diagrams and talk in the abstract, I want to really have an example project that we can implement and re-implement using different patterns. This should really drive home the differences between the patterns.

For this we'll be writing a server that speaks a subset of FTP. Why a subset? Because I want the focus of this section to be on the architecture pattern, not the protocol implementation. Why FTP? Because then we can test it without having to write our own clients. Lots of FTP clients already exist.

For the unfamiliar, FTP is the File Transfer Protocol. It defines a text-based protocol, typically spoken over TCP, for transferring files between two computers.

As you'll see, it feels a bit like browsing a filesystem. FTP makes use of simultaneous TCP sockets. One 'control' socket is used for sending FTP commands and their arguments between server and client. Each time that a file transfer is to be made, a new TCP socket is used. It's a nice hack that allows for FTP commands to continue to be processed on the control socket while a transfer is in progress.

Here's the protocol implementation of our FTP server. It defines some common methods for writing FTP formatted responses and building the control socket. It also provides a `CommandHandler` class that encapsulates the handling of individual commands on a per-connection basis. This is important. Individual connections on the same server may have different working directories, and this class honours that.

```
# ./code/ftp/common.rb
module FTP
  module Common
    CRLF = "\r\n"

    def initialize(port = 21)
      @control_socket = TCPServer.new(port)
      trap(:INT) { exit }
    end

    def respond(response)
      @client.write(response)
      @client.write(CRLF)
    end

    class CommandHandler
      attr_reader :connection
      def initialize(connection)
        @connection = connection
      end

      def pwd
        @pwd || Dir.pwd
      end

      def handle(data)
        cmd = data[0..3].strip.upcase
        options = data[4..-1].strip

        case cmd
        when 'USER'
          # Accept any username anonymously
          "230 Logged in anonymously"
      when 'SYST'
        # what's your name?
        "215 UNIX Working With FTP"

      when 'CWD'
        if File.directory?(options)
          @pwd = options
          "250 directory changed to #{pwd}"
        else
          "550 directory not found"
        end

      when 'PWD'
        "257 \"#{pwd}\" is the current directory"

      when 'PORT'
        parts = options.split(',')
        ip_address = parts[0..3].join('.')
        port = Integer(parts[4]) * 256 + Integer(parts[5])

        @data_socket = TCPSocket.new(ip_address, port)
        "200 Active connection established (#{port})"

      when 'RETR'
        file = File.open(File.join(pwd, options), 'r')
        connection.respond "125 Data transfer starting #{file.size} bytes"

        bytes = IO.copy_stream(file, @data_socket)
        @data_socket.close

        "226 Closing data connection, sent #{bytes} bytes"

      when 'LIST'
```

```
              connection.respond "125 Opening data connection for file list"

              result = Dir.entries(pwd).join(CRLF)
              @data_socket.write(result)
              @data_socket.close

              "226 Closing data connection, sent #{result.size} bytes"

            when 'QUIT'
              "221 Ciao"

            else
              "502 Don't know how to respond to #{cmd}"
            end
          end
        end
      end
    end
```

Did you see the `respond` method? It writes the response out on the connection. FTP uses a `\r\n` to signify message boundaries, just like HTTP.

This protocol implementation doesn't say much about networking or concurrency; that's the part we get to play with in the following chapters.

Chapter 19

Serial

The first network architecture pattern we'll look at is a *Serial* model of processing requests. We'll proceed from the perspective of our FTP server.

Explanation

With a serial architecture all client connections are handled serially. Since there is no concurrency, multiple clients are never served simultaneously.

The flow of this architecture is straightforward:

1. Client connects.
2. Client/server exchange requests and responses.
3. Client disconnects.
4. Back to step #1.

Implementation

```ruby
# ./code/ftp/arch/serial.rb
require 'socket'
require_relative '../common'

module FTP
  class Serial
    include Common

    def run
      loop do
        @client = @control_socket.accept
        respond "220 OHAI"

        handler = CommandHandler.new(self)

        loop do
          request = @client.gets(CRLF)

          if request
            respond handler.handle(request)
          else
            @client.close
            break
          end
        end
      end
    end
  end
end

server = FTP::Serial.new(4481)
server.run
```

Notice that this class is only responsible for networking and concurrency; it hands off the protocol handling to the `Common` module methods. It's a pattern you'll keep seeing. Let's take it from the top.

```ruby
# ./code/ftp/arch/serial.rb
  def run
    loop do
      @client = @control_socket.accept
      respond "220 OHAI"

      handler = CommandHandler.new(self)
```

This is the entry point for the server. As you can see, all of the logic happens inside a main outer loop.

The *only* call to `accept` inside this loop is the one you see at the top here. It accepts a connection from the `@control_socket` initialized in `Common`. The 220 response is a protocol implementation detail. FTP requires us to say 'hi' after accepting a new client connection.

The last bit here is the initialization of a `CommandHandler` for this connection. This class encapsulates the current state (current working directory) of the server on a per-connection basis. We'll feed the incoming requests to the `handler` object and get back the proper responses.

This bit of code is the concurrency blocker in this pattern. Because the server does *not* continue to accept connections while it's processing this one, there can be no concurrency. This difference will become more apparent as we look at how other patterns handle this.

```ruby
# ./code/ftp/arch/serial.rb
loop do
  request = @client.gets(CRLF)

  if request
    respond handler.handle(request)
  else
    @client.close
```

This rounds out the serial implementation of our FTP server.

It enters an inner loop where it `gets` requests from the client socket passing in the explicit separator. It then passes those requests to the `handler` which crafts the proper response for the client.

Given that this is a fully functioning FTP server (albeit, it only supports a subset of FTP), we can actually run the server and hook it up with a standard FTP client to see it in action:

```
$ ruby code/ftp/arch/serial.rb
```

```
$ ftp -a -v 127.0.0.1 4481
cd /var/log
pwd
get kernel.log
```

Considerations

It's hard to nail down the pros and cons for each pattern because it depends *entirely* on your needs. I'll do my best to explain where each pattern excels and what tradeoffs it makes.

The greatest advantage that a serial architecture offers is simplicity. There's no locks, no shared state, no way to confuse one connection with another. This also goes for resource usage: one instance handling one connection won't consume as many resources as many instances or many connections.

The obvious disadvantage is that there's no concurrency. Pending connections aren't processed even when the current connection is idle. Similarly, if a connection is using a slow link or pausing between sending requests the server remains blocked until that connection is closed.

This serial implementation is really just a baseline for the more interesting patterns that follow.

Chapter 20

Process per connection

This is the first network architecture we'll look at that allows parallel processing of requests.

Explanation

This particular architecture requires very few changes from the serial architecture in order to add concurrency. The code that accepts connections will remain the same, as will the code that consumes data from the socket.

The relevant change is that after accepting a connection, the server will `fork` a child process whose sole purpose will be the handling of that new connection. The child process handles the connection, then exits.

Forking Basics

Any time that you start up a program using `$ ruby myapp.rb`, for instance, a new Ruby process is spawned that loads and executes your code.

If you do a `fork` as part of your program you can actually create a new process *at runtime*. The effect of a `fork` is that you end up with two processes that are exact copies. The newly created process is considered the child; the original considered the parent. Once the `fork` is complete then you have two processes that can go their separate ways and do whatever they need to do.

This is tremendously useful because it means we can, for instance, `accept` a connection, `fork` a child process, and that child process automatically gets a copy of the client connection. Hence there's no extra setup, sharing of data, or locking required to start parallel processing.

Let's make the flow of events crystal clear:

1. A connection comes in to the server.
2. The main server process accepts the connection.
3. It forks a new child process which is an exact copy of the server process.
4. The child process continues to handle its connection in parallel while the server process goes back to step #1.

Thanks to kernel semantics these processes are running in parallel. While the new child process is handling the connection, the original parent process can continue to accept new connections and fork new child processes to handle them.

At any given time there will always be a single parent process waiting to accept connections. There may also be multiple child processes handling individual connections.

Implementation

```
# ./code/ftp/arch/process_per_connection.rb
require 'socket'
require_relative '../common'

module FTP
  class ProcessPerConnection
    include Common

    def run
      loop do
        @client = @control_socket.accept

        pid = fork do
          respond "220 OHAI"

          handler = CommandHandler.new(self)

          loop do
            request = @client.gets(CRLF)

            if request
              respond handler.handle(request)
            else
              @client.close
              break
            end
          end
        end

        Process.detach(pid)
      end
    end
  end
end
```

```
server = FTP::ProcessPerConnection.new(4481)
server.run
```

As you can see the majority of the code remains the same. The main difference is that the inner loop is wrapped in a call to `fork`.

```
# ./code/ftp/arch/process_per_connection.rb
@client = @control_socket.accept

pid = fork do
  respond "220 OHAI"

  handler = CommandHandler.new(self)
```

Immediately after `accept`ing a connection the server process calls `fork` with a block. The new child process will evaluate that block and then exit.

This means that each incoming connection gets handled by a single, independent process. The parent process will not evaluate the code in the block; it just continues along the execution path.

```
# ./code/ftp/arch/process_per_connection.rb
Process.detach(pid)
```

Notice the call to `Process.detach` at the end? After a process exits it isn't fully cleaned up until its parent asks for its exit status. In this case we don't care what the child exit status is, so we can detach from the process early on to ensure that its resources are fully cleaned up when it exits [2].

Considerations

This pattern has several advantages. The first is simplicity. Notice that very little extra code was required on top of the serial implementation in order to be able to service multiple clients in parallel.

A second advantage is that this kind of parallelism requires very little cognitive overhead. I mentioned earlier that `fork` effectively provides copies of everything a child process might need. There are no edge cases to look out for, no locks or race conditions, just simple separation.

An obvious disadvantage to this pattern is that there's no upper bound on the number of child processes it's willing to `fork`. For a small number of clients this won't be an issue, but if you're spawning dozens or hundreds of processes then your system will quickly fall over. This concern can be solved using the *Preforking* pattern discussed a few chapters from now.

Depending on your operating environment, the very fact that it uses `fork` might be an issue. `fork` is only supported on Unix systems. This means it's *not* supported on Windows or JRuby.

Another concern is the issue of using processes versus using threads. I'll save this discussion for the next chapter when we actually get to see threads.

Examples

- shotgun

[2] If you want to learn more about process spawning and zombie processes then you should get my other book Working With Unix Processes.

- inetd

Chapter 21

Thread per connection

Explanation

This pattern is *very* similar to the *Process Per Connection* pattern from the last chapter. The difference? Spawn a thread instead of spawning a process.

Threads vs. Processes

Threads and processes both offer parallel execution, but in very different ways. Neither is ever a silver bullet and your choice of which to use depends on your use case.

Spawning. When it comes to spawning, threads are much cheaper to spawn. Spawning a process creates a copy of everything the original process had. Threads are per-process, so if you have multiple threads they'll all be in the same process. Since threads share memory, rather than copying it, they can spawn much faster.

Synchronizing. Since threads share memory you must take great care when working with data structures that will be accessed by multiple threads. This usually means mutexes, locks, and synchronizing access between threads. Processes need none of this because each process has its own copy of everything.

Parallelism. Both offer parallel computation implemented by the kernel. One important thing to note about parallel threads in MRI is that the interpreter uses a *global* lock around the current execution context. Since threads are per-process they'll all be running inside the same interpreter. Even when using multiple threads MRI prevents them from achieving true parallelism. This isn't true in alternate Ruby implementations like JRuby or Rubinius 2.0.

Processes don't have this issue because each time a copy is made, the new process also gets its own copy of the Ruby interpreter, hence there's no global lock. In MRI, only processes offer true concurrency.

One more thing about parallelism and threads. Even though MRI uses a global interpreter lock it's pretty smart with threads. I mentioned in the *DNS* chapter that Ruby will allow other threads to execute while a given thread is blocking on IO.

In the end threads are lighter-weight; processes are heavier. Both offer parallel execution. Both have their use cases.

Implementation

```
                                                # ./code/ftp/arch/thread_per_connection.rb
require 'socket'
require 'thread'
require_relative '../common'

module FTP
  class ProcessPerConnection
    include Common

    def run
      Thread.abort_on_exception = true

      loop do
        @client = @control_socket.accept

        Thread.new do
          respond "220 OHAI"

          handler = CommandHandler.new(self)

          loop do
            request = @client.gets(CRLF)

            if request
              respond handler.handle(request)
            else
              @client.close
              break
            end
          end
        end
      end
    end
  end
```

```
end

server = FTP::ProcessPerConnection.new(4481)
server.run
```

This code is even closer to the serial pattern because there's no need to reap or detach a thread after spawning it. Once it's finished execution, it's picked up by the garbage collector.

Considerations

This pattern shares many of the same advantages as the previous one: very little code changes were required, very little cognitive overhead added. Although using threads can introduce issues with locking and synchronization, we don't have to worry about any of that here because each connection is handled by a single, independent thread.

One advantage of this pattern over *Process Per Connection* is that threads are lighter on resources, hence there can be more of them. This pattern should afford you more concurrency to service clients than you can have using processes.

But wait, remember that the MRI GIL comes into play here to prevent that from becoming a reality. In the end, neither pattern is a silver bullet. Each should be considered, tried, and tested.

This pattern shares a disadvantage with *Process Per Connection*: the number of threads can grow and grow until the system is overwhelmed. If your server is handling an increased number of connections, then your system could possibly be overwhelmed trying to maintain and switch between all the active threads. This can be resolved by limiting the number of active threads. We'll see this when we look at the *Thread Pool* pattern.

Examples

- WEBrick
- Mongrel

Chapter 22

Preforking

Explanation

This pattern harks back to the *Process Per Connection* architecture we saw a few chapters back.

This one also leans on processes as its means of parallelism, but rather than forking a child process for each incoming connection, it forks a bunch of processes when the server boots up, before any connections arrive.

Let's review the workflow:

1. Main server process creates a listening socket.
2. Main server process forks a horde of child processes.
3. Each child process accepts connections on the shared socket and handles them independently.
4. Main server process keeps an eye on the child processes.

The important concept is that the main server process opens the listening socket, but doesn't accept connections on it. It then forks a predefined number of child processes, each of will have a copy of the listening socket. The child processes then each call `accept` on the listening socket, taking the parent process out of the equation.

The best part about this is that we don't have to worry about load balancing or synchronizing connections across our child processes because the kernel handles that for us. Given more than one process trying to `accept` a connection on different copies of the same socket, the kernel balances the load and ensures that one, and only one, copy of the socket will be able to `accept` any particular connection.

Implementation

```ruby
# ./code/ftp/arch/preforking.rb
require 'socket'
require_relative '../common'

module FTP
  class Preforking
    include Common

    CONCURRENCY = 4

    def run
      child_pids = []

      CONCURRENCY.times do
        child_pids << spawn_child
      end

      trap(:INT) {
        child_pids.each do |cpid|
          begin
            Process.kill(:INT, cpid)
          rescue Errno::ESRCH
          end
        end

        exit
      }

      loop do
        pid = Process.wait
        $stderr.puts "Process #{pid} quit unexpectedly"

        child_pids.delete(pid)
        child_pids << spawn_child
      end
    end

    def spawn_child
      fork do
        loop do
          @client = @control_socket.accept
          respond "220 OHAI"

          handler = CommandHandler.new(self)

          loop do
            request = @client.gets(CRLF)

            if request
              respond handler.handle(request)
            else
              @client.close
              break
            end
          end
        end
      end
    end
  end
end

server = FTP::Preforking.new(4481)
server.run
```

This implementation is notably different from the three we've looked at thus far. Let's talk about in two chunks, starting at the top.

```ruby
# ./code/ftp/arch/preforking.rb
CONCURRENCY = 4

def run
  child_pids = []

  CONCURRENCY.times do
    child_pids << spawn_child
  end

  trap(:INT) {
    child_pids.each do |cpid|
      begin
        Process.kill(:INT, cpid)
      rescue Errno::ESRCH
      end
    end

    exit
  }

  loop do
    pid = Process.wait
    $stderr.puts "Process #{pid} quit unexpectedly"

    child_pids.delete(pid)
    child_pids << spawn_child
  end
end
```

This method begins by invoking the `spawn_child` method a number of times, based on the number stored in `CONCURRENCY`. The `spawn_child` method (more on it below) will actually `fork` a new process and return its unique process id (pid).

After spawning the children, the parent process defines a signal handler for the `INT` signal. This is the signal that your process receives when you type `Ctrl-C`, for instance. This bit of code simply forwards an `INT` signal received by the parent to its child processes. Remember that the child processes exist independently of the parent and are happy to live on even if the parent process dies. As such, it's important for a parent process to clean up their child processes before exiting.

After signal handling, the parent process enters a loop around `Process.wait`. This method will block until a child process exits. It returns the pid of the exited child. Since there's no reason for the child processes to exit, we assume it's an anomaly. We print a message on STDERR and spawn a new child to take its place.

Some preforking servers, notably Unicorn [2], have the parent process take a more active role in monitoring its children. For example, the parent may look to see if any of the children are taking a long time to process a request. In that case the parent process will forcefully kill the child process and spawn a new one in its place.

[2] http://unicorn.bogomips.org

```ruby
# ./code/ftp/arch/preforking.rb
def spawn_child
  fork do
    loop do
      @client = @control_socket.accept
      respond "220 OHAI"

      handler = CommandHandler.new(self)

      loop do
        request = @client.gets(CRLF)

        if request
          respond handler.handle(request)
        else
          @client.close
          break
        end
      end
    end
  end
end
```

The core of this method should be familiar. This time it's wrapped in a `fork` and a `loop`. So a new child process is forked *before* calling `accept`. The outermost loop ensures that as each connection is handled and closed, a new connection is handled. In this way each child process will be in its own accept loop.

Considerations

There are several things at play that make this a great pattern.

Compared to the similar *Process Per Connection* architecture, *Preforking* doesn't have to pay the cost of doing a `fork` during each connection. Forking a process isn't a cheap operation, and in *Process Per Connection*, every single connection must begin with paying that cost.

As hinted earlier, this pattern prevents too many processes from being spawned, because they're all spawned beforehand.

One advantage that this pattern has over a similar threaded pattern is complete separation. Since each process has its own copy of everything, including the Ruby interpreter, a failure in one process will not affect any other processes. Since threads share the same process and memory space, a failure in one thread may affect other threads unpredictably.

A disadvantage of using *Preforking* is that forking more processes means that your server will consume more memory. Processes don't come cheap. Given that each forked process gets a copy of *everything*, you can expect your memory usage to increase by up to 100% of the parent process size on each fork.

In this way a 100MB process will occupy 500MB after forking four children. And this would allow only 4 concurrent connections.

I won't harp this point too much here, but this code is really simple. There are a few concepts that need to be understood, but overall it's simple, with little to worry about in the way of things going awry at runtime.

Examples

- Unicorn

Chapter 23

Thread Pool

Overview

This pattern is to *Preforking* what *Thread Per Connection* is to *Process Per Connection*. Much like *Preforking*, this pattern will spawn a number of threads when the server boots and defer connection handling to each independent thread.

The flow of this architecture is the same as the previous, but substitute 'thread' for 'process'.

Implementation

```ruby
# ./code/ftp/arch/thread_pool.rb
require 'socket'
require 'thread'
require_relative '../common'

module FTP
  class ThreadPool
    include Common

    CONCURRENCY = 25

    def run
      Thread.abort_on_exception = true
      threads = ThreadGroup.new

      CONCURRENCY.times do
        threads.add spawn_thread
      end

      sleep
    end

    def spawn_thread
      Thread.new do
        loop do
          @client = @control_socket.accept
          respond "220 OHAI"

          handler = CommandHandler.new(self)

          loop do
            request = @client.gets(CRLF)

            if request
```

```
              respond handler.handle(request)
            else
              @client.close
              break
            end
          end
        end
      end
    end
  end
end

server = FTP::ThreadPool.new(4481)
server.run
```

Again, two methods here. One spawns the threads, the other encapsulates the spawning and thread behaviour.

```
# ./code/ftp/arch/thread_pool.rb
CONCURRENCY = 25

def run
  Thread.abort_on_exception = true
  threads = ThreadGroup.new

  CONCURRENCY.times do
    threads.add spawn_thread
  end

  sleep
```

This method creates a `ThreadGroup` to keep track of all the threads. `ThreadGroup` is a bit like a thread-aware Array. You add threads to the `ThreadGroup`, but when a member thread finishes execution it's silently dropped from the group.

You can use `ThreadGroup#list` to get a list of all the threads currently in the group, all of which will be active. We don't actually use this in this implementation but `ThreadGroup` would be useful if we wanted to act on all active threads (to `join` them, for instance).

Much like in the last chapter, we simply call the `spawn_thread` method as many times as `CONCURRENCY` calls for. Notice how the `CONCURRENCY` number is higher here than in *Preforking*? Again, that's because threads are lighter weight and, therefore, we can have more of them. Just keep in mind that the MRI GIL mitigates some of this gain.

The end of this method calls `sleep` to prevent it from exiting. The main thread remains idle while the pool does the work. Theoretically it could be doing its own work monitoring the pool, but here it just `sleep`s to prevent it from exiting.

```
# ./code/ftp/arch/thread_pool.rb
  def spawn_thread
    Thread.new do
      loop do
        @client = @control_socket.accept
        respond "220 OHAI"

        handler = CommandHandler.new(self)

        loop do
          request = @client.gets(CRLF)

          if request
            respond handler.handle(request)
          else
            @client.close
            break
          end
        end
      end
```

This method is pretty unexciting. It follows the same pattern as *Preforking*. Namely, spawn a thread that loops around the connection handling code. Again, the kernel ensures that a single connection can only be `accept`ed into a single thread.

Considerations

Much of the considerations of this pattern are shared with the previous.

Besides the obvious thread vs. process tradeoff this pattern will not need to spawn threads each time it handles a connection, does not have any crazy locks or race conditions, yet still provides parallel processing.

Examples

- Puma

Chapter 24

Evented (Reactor)

Up until now all of the patterns we've seen have really been a variation on the serial pattern. Besides the serial pattern itself, the other patterns used the same structure but wrapped threads or processes around it.

This pattern takes a whole different approach that won't look anything like the others.

Overview

The evented pattern (based on the Reactor pattern [2]) seems to be all the rage these days. It's at the core of libraries like EventMachine, Twisted, Node.js, Nginx, and others.

This pattern is single-threaded and single-process, yet it can achieve levels of concurrency at least on par with the other patterns discussed.

It centers around a central connection multiplexer (hereby referred to as the Reactor core). Each stage of the connection lifecycle is broken down into individual events that can be interleaved and handled in any given order. The different stages of a connection are simply the possible IO operations: accept, read, write, close.

The central multiplexer monitors all the active connections for events and dispatches the relevant code upon being triggered by an event.

[2] http://en.wikipedia.org/wiki/Reactor_pattern

Let's review the workflow:

1. The server monitors the listening socket for incoming connections.
2. Upon receiving a new connection it adds it to the list of sockets to monitor.
3. The server now monitors the active connection as well as the listening socket.
4. Upon being notified that the active connection is readable the server reads a chunk of data from that connection and dispatches the relevant callback.
5. Upon being notified that the active connection is *still* readable the server reads another chunk and dispatches the callback again.
6. The server receives another new connection; it adds that to the list of sockets to monitor.
7. The server is notified that the first connection is ready for writing, so the response is written out on that connection.

Keep in mind that all of this is happening in a single thread. Notice that the server was able to `accept` a new connection while the first connection was still in the middle of its read/write flow?

The server is simply splitting each operation into small chunks so that the various events pertaining to multiple connections can be interleaved.

Time to dig into the code.

Implementation

```
# ./code/ftp/arch/evented.rb
require 'socket'
require_relative '../common'

module FTP
  class Evented
    CHUNK_SIZE = 1024 * 16
    include Common

    class Connection
      include Common

      attr_reader :client

      def initialize(io)
        @client = io
        @request, @response = "", ""
        @handler = CommandHandler.new(self)

        @response = "220 OHAI" + CRLF
        on_writable
      end

      def on_data(data)
        @request << data

        if @request.end_with?(CRLF)
          # Request is completed.
          @response = @handler.handle(@request) + CRLF
          @request = ""
        end
      end

      def on_writable
```

```
        bytes = client.write_nonblock(@response)
        @response.slice!(0, bytes)
      end

      def monitor_for_reading?
        true
      end

      def monitor_for_writing?
        !(@response.empty?)
      end
    end

    def run
      @handles = {}

      loop do
        to_read = @handles.values.select(&:monitor_for_reading?).map(&:client)
        to_write = @handles.values.select(&:monitor_for_writing?).map(&:client)

        readables, writables = IO.select(to_read + [@control_socket], to_write)

        readables.each do |socket|
          if socket == @control_socket
            io = @control_socket.accept
            connection = Connection.new(io)
            @handles[io.fileno] = connection

          else
            connection = @handles[socket.fileno]

            begin
              data = socket.read_nonblock(CHUNK_SIZE)
              connection.on_data(data)
            rescue Errno::EAGAIN
            rescue EOFError
              @handles.delete(socket.fileno)
            end
          end
        end

        writables.each do |socket|
          connection = @handles[socket.fileno]
          connection.on_writable
        end
      end
    end
  end
end

server = FTP::Evented.new(4481)
server.run
```

You can see already that this implementation follows a different cadence than the others that we've looked at thus far. Let's start breaking it down by section.

```
                                                        # ./code/ftp/arch/evented.rb
class Connection
```

This bit of code defines a `Connection` class for our *Evented* server. This is the first pattern that's actually defined a `Connection` class to represent an individual connection. Why is this needed?

All of the other patterns used processes/threads to separate connections from each other. No matter which way they were using processes/threads, they always made sure that a single, independent process/thread only handled a single connection at any given time. So each connection was more-or-less represented by a process/thread.

The *Evented* pattern is single-threaded so it needs to represent each independent connection with an object so they don't end up trampling on each other's state.

```ruby
# ./code/ftp/arch/evented.rb
class Connection
  include Common

  attr_reader :client

  def initialize(io)
    @client = io
    @request, @response = "", ""
    @handler = CommandHandler.new(self)

    @response = "220 OHAI" + CRLF
    on_writable
  end
```

Starting at the top of the `Connection` class, we see some familiarity.

The connection stores the actual underlying `IO` object in its `@client` instance variable and makes that accessible to the outside world with an `attr_accessor`.

When an individual connection is initialized it gets its own `CommandHandler` instance, just as before. After that it writes out the customary 'hello' response that FTP requires. However, rather than writing it out to the client connection directly, it just assigns the response body to the `@response` variable. As we'll see in the next section this triggers the Reactor to take over and send this data out to the client.

```ruby
# ./code/ftp/arch/evented.rb
  def on_data(data)
    @request << data

    if @request.end_with?(CRLF)
      # Request is completed.
      @response = @handler.handle(@request) + CRLF
      @request = ""
    end
  end

  def on_writable
    bytes = client.write_nonblock(@response)
    @response.slice!(0, bytes)
  end

  def monitor_for_reading?
    true
  end

  def monitor_for_writing?
    !(@response.empty?)
  end
end
```

This part of `Connection` defines the lifecycle methods that the Reactor core interacts with.

For example, when the Reactor reads data from the client connection it triggers the `on_data` with that data. Inside that method it checks to see if it's received a complete request. If it has then it asks the `@handler` to build the response and, once again, assigns that to `@response`.

The `on_writable` method is called when the client connection is ready to be written to. This is where the `@response` variable is dealt with. It writes what it can from the `@response` out to the client connection. Based on how many bytes it was able to write, it slices the `@response` to remove the bit that was successfully written.

As such, any subsequent writes will only write the part of the `@response` that wasn't able to be written this time around. If the whole thing was able to be written, the `@response` will be sliced to an empty string, and nothing more will be written.

The last two methods, `monitor_for_reading?` and `monitor_for_writing?`, are queried by the Reactor to see if it should monitor the state of this particular connection for reading, writing, or both. In this case we're always willing to read new data if it's available, but we only want to monitor for the ability to write if there's a `@response` to be written. Given an empty `@response`, the Reactor won't notify us if the client connection is writable.

```
                                                # ./code/ftp/arch/evented.rb
    def run
      @handles = {}

      loop do
        to_read = @handles.values.select(&:monitor_for_reading?).map(&:client)
        to_write = @handles.values.select(&:monitor_for_writing?).map(&:client)

        readables, writables = IO.select(to_read + [@control_socket], to_write)
```

This is the main work of the Reactor core.

The `@handles` Hash looks something `{6 => #<FTP::Evented::Connection:xyz123>}` where the keys are file descriptor numbers and the values are `Connection` objects.

The first lines inside the main loop ask each of the active connections if they want to be monitored for reading or writing, using the lifecycle methods we saw earlier. It grabs a reference to the underlying `IO` object for each of the eligible connections.

The Reactor then passes these `IO` instances to `IO.select` with no timeout. This `IO.select` call will block until one of the monitored sockets gets an event that requires attention.

Note that the Reactor also sneaks the `@control_socket` into the connections to monitor for reading so it can detect new incoming client connections.

```
# ./code/ftp/arch/evented.rb
      readables.each do |socket|
        if socket == @control_socket
          io = @control_socket.accept
          connection = Connection.new(io)
          @handles[io.fileno] = connection

        else
          connection = @handles[socket.fileno]

          begin
            data = socket.read_nonblock(CHUNK_SIZE)
            connection.on_data(data)
          rescue Errno::EAGAIN
          rescue EOFError
            @handles.delete(socket.fileno)
          end
        end
      end

      writables.each do |socket|
        connection = @handles[socket.fileno]
        connection.on_writable
      end
    end
  end
```

This is the part of the Reactor that triggers appropriate methods based on events it receives from `IO.select`.

First, it handles the sockets deemed 'readable'. If the `@control_socket` was readable this means that there's a new client connection. So the Reactor `accept`s, builds a new

`Connection` and slots it into the `@handles` Hash so it can be monitored the next time around the loop.

Next, it handles the case where a socket deemed 'readable' was a regular client connection. In this case it attempts to read the data and trigger the `on_data` method of the appropriate `Connection`. In the case that the read would block (`Errno::EAGAIN`), it doesn't do anything special, just lets the event fall through. In the case that the client disconnected (`EOFError`), it makes sure to remove the entry from the `@handles` Hash so the appropriate objects can be garbage collected and will no longer be monitored.

The last bit handles sockets deemed 'writable' simply by triggering the `on_writable` method of the appropriate `Connection`.

Considerations

This pattern is notably different than the others and, as such, produces notably different advantages and disadvantages.

First of all, this pattern has a reputation of being able to handle extremely high levels of concurrency, numbering in the thousands, or tens of thousands, of concurrent connections. This is something that the other patterns simply can't approach because they're limited by processes/threads.

If your server attempts to spawn 5000 threads to handle 5000 connections then things will likely grind to a halt. This pattern wins, hands down, in terms of handling concurrent connections.

The main disadvantage of this pattern is the programming model that it forces upon you. On the one hand the model is simpler because there are no processes/threads to deal with. This means there are no issues of shared memory, synchronization, runaway processes, etc. to deal with. However, given that all this concurrency is happening inside a single thread, there's one very important rule to follow: never block the Reactor.

To illustrate this, let's use an example: imagine that our example FTP server retrieved files from a remote server instead of retrieving files from the local filesystem. When the Reactor notified our `Connection` that a request was available to read, it would read the request and build the response.

But now, building the response entails fetching data from a remote server and streaming it back to the client. Imagine that this remote server was on a slow link that led to a slow connection for us to stream the remote data. What effect would this have on the Reactor?

Given that everything runs in same thread, this slow outgoing connection would block the whole Reactor! When the Reactor triggers a method on a `Connection`, the whole Reactor is blocked until that method returns. In the meantime, no other data is being read, no new connections are being accepted, etc.

As such it's very important that anything that your application code wants to do be done very quickly. So how should we handle a slow outgoing connection with a Reactor? Use the Reactor itself!

If you're using this pattern you need to make sure that any blocking IO is handled by the Reactor itself. In our example this would mean the slow remote connection would need to be encapsulated inside its own subclass of `Connection` that defined its own `on_data` and `on_writable` methods.

When the Reactor is able to read some data from that slow connection, it would then trigger the appropriate `on_data` method, which would set up a response to be written out to the client connection. In this way the Reactor can continue processing other connections while waiting for this slow remote connection, yet still handle that connection when it's ready.

In short, this pattern offers some obvious advantages and really simplifies some aspects of socket programming. On the other hand, it requires you to rethink all of the IO that your app does. It's easy to cancel all of the offered benefits with a bit of slow code or some third-party library that does blocking IO.

Examples

- EventMachine
- Celluloid::IO
- Twisted

Chapter 25

Hybrids

This is the last part of the network patterns section of the book. It doesn't cover a specific pattern itself, but rather the concept of making a hybrid pattern that uses one or more of the patterns described in this section.

Although any of these architectures can be applied to any kind of service (we saw FTP in the previous chapters), there's been a lot of attention given to HTTP servers in modern times. This is unsurprising given the prevalence of the web. The Ruby community is at the forefront of this web movement and has its fair share of different HTTP servers to choose from. Hence, the real-world examples we'll look at in this chapter are all HTTP servers.

Let's dive in to some examples.

nginx

The nginx [2] project provides an extremely high-performance network server written in C. Indeed, its web site claims it can serve *1 million concurrent requests* on a single server. nginx is often used in the Ruby world as an HTTP proxy in front of web application servers, but it can speak HTTP, SMTP, and others.

So how does nginx achieve this kind of concurrency?

[2] http://nginx.org

At its core [3], nginx uses the *Preforking* pattern. However, inside each forked process is the *Evented* pattern. This makes a lot of sense as a high-performance choice for a few reasons.

First, all of the spawning costs are paid at boot time when nginx forks child processes. This ensures that nginx can take maximum advantage of multiple cores and server resources. Second, the *Evented* pattern is notable in that it doesn't spawn anything and doesn't use threads. One issue when using threads is the overhead required for the kernel to manage and switch context between all of the active threads.

nginx is packed with tons of other features that make it blazing fast, including tight memory management that can only be accomplished in a language like C, but at its core it uses a hybrid of the patterns described in the last few chapters.

Puma

The puma rubygem provides "a Ruby web server built for concurrency" [4]. Puma is designed as the go-to HTTP server for Ruby implementations *without* a GIL (Rubinius or JRuby) because it leans heavily on threads. The Puma README [5] provides a good overview of where it's applicable and reminds us about the effect of the GIL on threading.

So how does Puma achieve its concurrency?

At a high level Puma uses a *Thread Pool* to provide concurrency. The main thread always `accept`s new connections and then queues up the connection to the thread pool for

[3] http://www.aosabook.org/en/nginx.html
[4] http://puma.io
[5] https://github.com/puma/puma#description

handling. This is the whole story for HTTP connections that don't use keep-alive [6]. But Puma does support HTTP keep-alive. When a connection is handled and its first request asks for the connection to be kept alive, Puma respects this and doesn't close it.

But now Puma can no longer just `accept` on that connection; it needs to monitor it for new requests and process those as well. It does this with an *Evented* reactor. When a new request arrives for a previously kept alive connection, that request is again queued up to the *Thread Pool* for handling.

So Puma's request handling is always done by a pool of threads. This is supported by a reactor that monitors any persistent connections.

Again, Puma is full of other optimizations, but at its core it's built on a hybrid of the patterns from the last few chapters.

EventMachine

EventMachine is well known in the Ruby world as an event-driven I/O library. It uses the *Reactor* pattern to provide high stability and scalability. Its internal are written in C but provide a Ruby interface as a C extension.

So how does EventMachine achieve its concurrency?

At its core EventMachine is an implementation of an *Evented* pattern. It's a single-threaded event loop that can handle network events on many concurrent connections. But EventMachine also provides a *Thread Pool* for deferring any long-running or blocking operations that would otherwise slow down the Reactor.

[6] http://en.wikipedia.org/wiki/HTTP_persistent_connection

EventMachine supports a ton of features, including the ability to monitor spawned processes, network protocol implementations and more. This example of using multiple architectures is just one way that it improves concurrency.

Chapter 26

Closing Thoughts

You now know the basics, and I daresay more, about socket programming. You can apply this stuff to Ruby, and to anywhere else you may find yourself programming. This is knowledge that will stick with you and remain useful.

Thanks for taking the time to read this book. I hope it gives you a deeper understanding of the work you do and the technologies you work with. My email address is jesse@jstorimer.com and I'm happy to chat about the book or any related programming topics.