Get it done.
One event at a time.

How I learned to stop worrying and love EventMachine.

Table of Contents

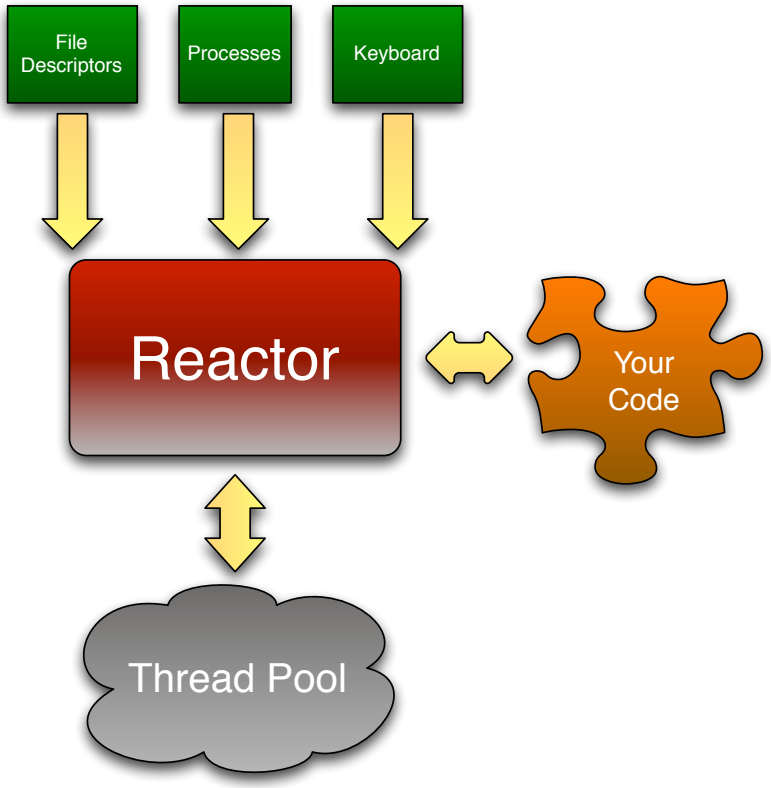
Introduction

So, I guess it would be useful, as a place to start, to know what we're getting ourselves into. What is EventMachine, and what can it do for me. Well, the first part of that is easy. EventMachine is a high performance implementation of the Reactor Pattern.[1]

Great, um, wait, what's the Reactor Pattern? According to Wikipedia:

> The reactor design pattern is a concurrent programming pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers.

Basically, EventMachine (EM) handles all that low level stuff including listening on sockets, creating network connections, handling timers and some concurrency primitives. EM provides a lot of the core functionality we need to create high performance API servers and services. EventMachine takes into account the challenges and lessons of the C10K problem[2] by Dan Kegel and others.



1 http://en.wikipedia.org/wiki/Reactor_pattern

2 http://www.kegel.com/c10k.html

Along with the networking capabilities, EM provides a thread pool which can be used to defer long running tasks to background threads. This keeps the application snappy and responsive. We all like responsive, right? Of course there are trade-offs involved in using the thread pool, mostly related to the green threads implementation in Ruby 1.8.

Let's take a look at a simple example to get started. We'll use the EM version of *Hello World*, the Echo server. Note, text in green is a reply from the server.

```
require 'eventmachine'

class Echo < EM::Connection
  def receive_data(data)
    send_data(data)
  end
end

EM.run do
  EM.start_server("0.0.0.0", 10000, Echo)
end
```

```
Rei:~ dj2$ telnet localhost 10000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
helo
helo
goodbye cruel world
goodbye cruel world
```

We'll get into the particulars later, but, at a high level we're starting a server on port 10000 that will use our *Echo* class when connections are established. When data is received on the port EM will execute the *receive_data* method and we just echo any data received back to the client by calling *send_data*.

In essence, you're associating an instance of the *Echo* class with a file descriptor. Whenever there is activity on the file descriptor the instance of your class will be called to handle the activity. This is the basis of most EventMachine programming, the reactor listens on a file description and executes some callbacks in an instance of your class.

The interesting thing to note is that our *Echo* class has no concept of any underlying network principles. There is no concept of packets or headers we just implement *Echo#receive_data* and we're able to retrieve network data.

Getting Started

With introductions out of the way, we can start on our merry path to EventMachine nirvana. There are a few basic tasks and commands that you'll be using as you forge ahead so we'll begin at the beginning, the basics.

To start, let's create the simplest, and possibly stupidest, EM application.

```
require 'eventmachine'

EventMachine::run
```

A fairly unexciting example, but serves as a good starting point. There are a couple of things to note. First, we require *eventmachine* in order to get access to EM. Second, for the astute reader, I switched from the first example using *EM.* to *EventMachine::*. I could also have done *EM::* or *EventMachine.*. All forms are equivalent when working with EventMachine. Personally, I prefer *EM.* as it's shorter and cleaner. Lastly, if you ran the example, you'll notice it never finished. When you call *EM#run* it will kick off the EventMachine reactor which will keep happy running until told to stop. Which we never actually did.

How do we stop this beastie you say? Easy enough.

```
require 'eventmachine'

EM.run do
  EM.add_timer(1) { EM.stop }
end
```

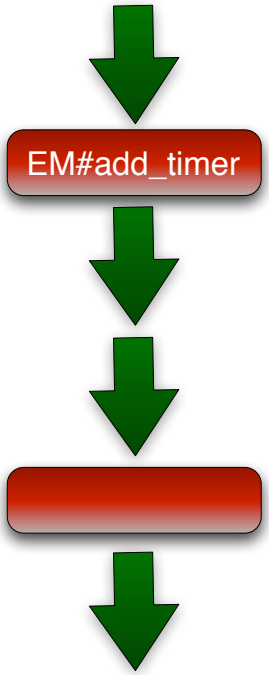
Here we're providing a block to *EM#run*. The block will be executed after the reactor is initialized but before the run loop is kicked off. You can put any initialization you need into this block. In this case, we're creating a timer and having it execute *EM#stop*. We could also have called *EM#stop_event_loop.* These two calls are equivalent and will both shutdown the reactor. Any code after your *EM#run* block will be executed after the reactor is terminated.

Timers

Our next stop on the tour is with EventMachine timers. There are two types of timers available. One shot timers and periodic timers. There are two different, but equivalent, ways of adding timers to EventMachine. The more common route is to use *EM#add_timer* and *EM#add_periodic_timer* as seen in the example below.

```
require 'eventmachine'

EM.run do
  EM.add_timer(5) do
    puts "BOOM"
    EM.stop_event_loop
  end
  EM.add_periodic_timer(1) do
    puts "Tick ... "
  end
end
```




```
titania:examples dj2$ ruby timer.rb
Tick...
Tick...
Tick...
Tick...
BOOM
```

This example adds two timers to the system. One periodic that will fire, at most, once a second, and a one-shot timer will fire in, at least, five seconds. The wording here is important. EM doesn't guarantee when the timer will fire, just the time frame in which it *may* fire.

The equivalent example, just using classes instead of methods, is below.

```
require 'eventmachine'

EM.run do
  EM::Timer.new(5) do
    puts "BOOM"
    EM.stop
  end
  EM::PeriodicTimer.new(1) do
    puts "Tick ..."
  end
end
```

In terms of functionality, these two examples are identical. I've seen the first way, using *EM#add_timer* and *EM#add_periodic_timer* used a lot more often then the class method.

There is one case that I know of where you *have* to use the class method. That's canceling periodic timers. You can cancel a one-shot timer using *EM#cancel_timer* and provide the signature returned from *EM#add_timer*. The problem with periodic timers is that they receive a new signature each time the timer is re-scheduled. Without knowing the signature you can't cancel the timer.

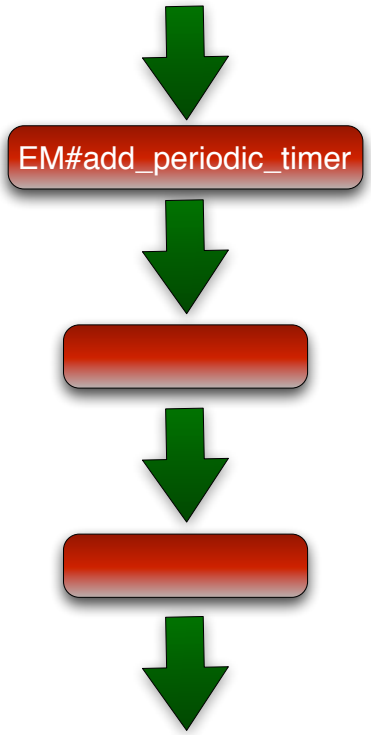
If you need to be able to cancel a periodic timer you'll need to use *EM::PeriodicTimer.* This will provide us with *EM::PeriodicTimer#cancel* which, you guessed it, allows us to cancel the periodic timer.

```
require 'eventmachine'

EM.run do
  p = EM::PeriodicTimer.new(1) do
    puts "Tick ..."
  end

  EM::Timer.new(5) do
    puts "BOOM"
    p.cancel
  end

  EM::Timer.new(8) do
    puts "The googles, they do nothing"
    EM.stop
  end
end
```




```
titania:examples dj2$ ruby timer_cancel.rb
Tick...
Tick...
Tick...
Tick...
BOOM
The googles, they do nothing
```

Deferring and Delaying Work

If you'll remember back to our definition of the Reactor Pattern, you'll note that the reactor is single threaded. EventMachine holds true to this single threaded approach to the reactor. The reactor itself is single threaded and the EM methods which work with the reactor are *not* thread-safe.

This has two outcomes. Firstly, we'll probably have code that takes a long time to run. Database queries, remote HTTP requests, etc. We need to be able to farm this code out to a background thread to be efficient. Secondly, once we move our code to a background thread, we need to be able to tell the reactor to do work for us.

This is where *EM#defer* and *EM#next_tick* come into play.

Using *EM#defer* we can schedule the execution of a block to one of the threads in EventMachines thread pool. This pool provides us with a fixed twenty threads. Great, we've got our code running in the background, how do we tell the client the results? If

needed, *EM#defer* takes a second parameter, the *callback*. This callback will be executed on the main reactor thread and will be provided with the return value of our deferred operation. We can use this callback to handle the client communications.

There is a downside to *EM#defer* which has to do with the Ruby 1.8 implementation of threads. Ruby doesn't have real operating system threads. It has what are called green threads. There is one operating system thread that everything runs on and Ruby creates its threads on top of this OS level thread. This means Ruby is handling all the thread scheduling and if anything takes the OS level thread your process will block. So, you need to be careful of what you're doing inside any threads you defer to make sure you're not blocking Ruby as this will block the world.

Using *EM#next_tick* we can schedule the execution of a block to happen on the next reactor run loop iteration. The execution will happen in the main reactor thread. For all intents and purposes you can consider this delay instantaneous, you're, essentially, scheduling the code to run on a different thread. Since EM is not thread safe, if we try to do *EM#defer* from a thread we can run into serious issues, and potentially crashes.

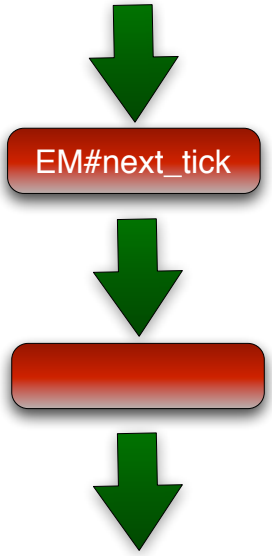
For example, if you're doing some work in a deferred thread and you need to establish a new outbound connection with *EM::HttpClient* you need to do this work on the main thread. We can place the connection creation inside the block provided to *EM#next_tick*.

EM#next_tick

Using *EM#next_tick* is very similar to working with *EM#add_timer.* We specify a block to *EM#next_tick* that will get executed on the next iteration of the reactor loop.

```
require 'eventmachine'

EM.run do
  EM.add_periodic_timer(1) do
    puts "Hai"
  end
  EM.add_timer(5) do
    EM.next_tick do
      EM.stop_event_loop
    end
  end
end
```




```
titania:examples dj2$ ruby next_tick.rb
Hai
Hai
Hai
Hai
```

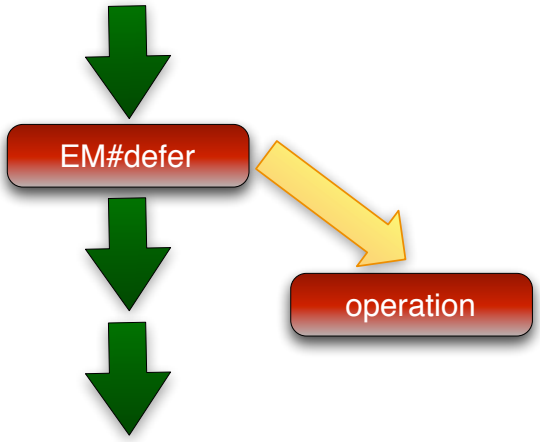
Everything scheduled using *EM#next_tick* will happen synchronously in the main thread. Any long running tasks inside an *EM#next_tick* block will cause the entire program to block until complete. Typically a bad thing.

EM#defer

While similar, using *EM#defer* has the added capability of providing a callback function that will be executed on the main thread after the operation has completed on the background thread. You do not have to provide the callback function.

```
require 'eventmachine'
require 'thread'

EM.run do
  EM.add_timer(2) do
    puts "Main #{Thread.current}"
    EM.stop_event_loop
  end
  EM.defer do
    puts "Defer #{Thread.current}"
  end
end
```



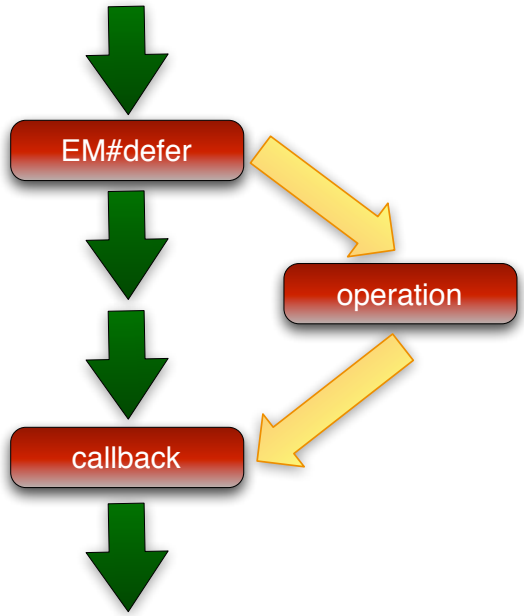

```
titania:examples dj2$ ruby defer.rb
Defer #<Thread:0x5637f4>
Main #<Thread:0x35700>
```

Providing a block to *EM#defer* will start executing the block on a background thread and continue on its merry way. Note, you should make sure that the code you execute on the thread won't occupy the thread forever as EventMachine won't detect this condition and the thread will be tied up indefinitely. The thread pool size is fixed, so if you start losing threads you won't get them back.

We can also utilize the ability for *EM#defer* to execute a callback function after running your operation. The callback function will be called with the return value of the operation if you specify the callback block to receive a parameter.

```
require 'eventmachine'

EM.run do
  op = proc do
    2 + 2
  end
  callback = proc do |count|
    puts "2 + 2 == #{count}"
    EM.stop
  end
  EM.defer(op, callback)
end
```




```
Rei:EventMachine dj2$ ruby defer_callback.rb
2 + 2 == 4
```

Here we're creating a *proc* that will sum two numbers. The result of this sum will be returned from the operation and passed to our callback. The callback will then be executed on the main reactor thread. The callback is outputting our result.

Lightweight Concurrency

EventMachine has two built-in mechanisms to handle light-weight concurrency[3]. They are, spawned processes and deferrables. Note, even though they're called spawned *processes* these are not operating system processes. The name comes from Erlang and can be a little confusing.

The main idea behind these two mechanisms is to be lighter, in CPU and memory, then regular Ruby threads. A lot of the work behind lightweight concurrency will need to be handled by your application. The code behind the deferrables and spawned processes will not be executed until your application specifically requests it to be executed.

Let's take a look at how these mechanisms work.

3 http://rubyeventmachine.com/browser/trunk/docs/LIGHTWEIGHT_CONCURRENCY

EM::Deferrable

We'll start by taking a look at *EM::Deferrable*[4]. When you mix *EM::Deferrable* into your class you are provided with the ability to associate callbacks and errbacks with instances of that class. You can define any number of callbacks and errbacks that may be executed. Callbacks and errbacks are executed in the order they were attached to the instance.

In order to cause the callbacks and errbacks to fire you call *#set_deferred_status* on the object. You'll provide either *:succeeded* or *:failed* as the parameter which will trigger the callbacks, if *:succeeded* was specified, or the errbacks, if *:failed* was specified. These blocks will be executed *immediately* on the main thread. There is also some syntactic sugar allowing you to call *#succeed* and *#fail* on the deferrable object to set the status.

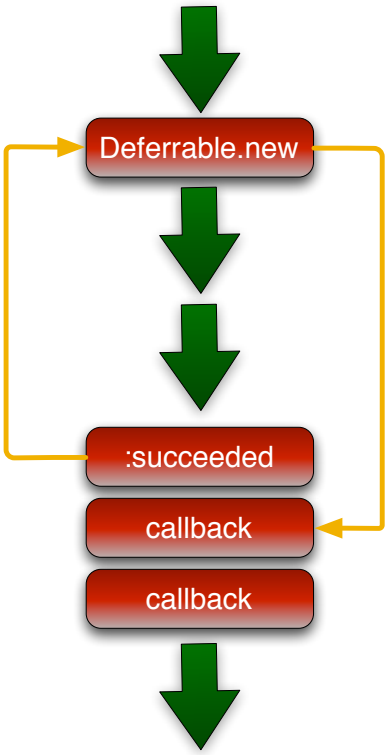
Once the status has been specified on the object, any future callbacks or errbacks will be executed *immediately* upon creation, as specified by the status.

Let's take a look at how this works in practice.

```
require 'eventmachine'

class MyDeferrable
  include EM::Deferrable
  def go(str)
    puts "Go #{str} go"
  end
end

EM.run do
  df = MyDeferrable.new
  df.callback do |x|
    df.go(x)
    EM.stop
  end
  EM.add_timer(1) do
    df.set_deferred_status :succeeded, "SpeedRacer"
  end
end
```




```
titania:EventMachine dj2$ ruby deferrable.rb
Go SpeedRacer go
```

4 http://rubyeventmachine.com/browser/trunk/docs/DEFERRABLES

Taking a closer look at what's going on here, we start by inserting *include EM::Deferrable* into our class. This will mixin the needed methods to make us a deferrable.

We create our *MyDeferrable* class as normal and proceed to call *#callback* and *#errback* on our instance. You can specify any number of parameters to the callback and errback blocks. When we have determined if our work has succeeded or failed we execute *#set_deferred_status* on the object. In this case we've *:succeeded* and, as we defined our callback block with one parameter, pass *"SpeedRacer*" to the callback.

By default, all callbacks will be executed with the same parameter. It's possible, inside the callback, to call *#set_deferred_status* again with a different set of parameters for subsequent callbacks to receive. This is further explained in the deferrable documentation.

There are times when you don't need to implement an entire class just to get the deferrable object properties. In these cases EventMachine provides you with *EM::DefaultDeferrable.* Everything works the same as mixing in *EM::Deferrable* except you just do *EM::DefaultDeferrable.new* instead of a custom class..

EM::SpawnedProcess

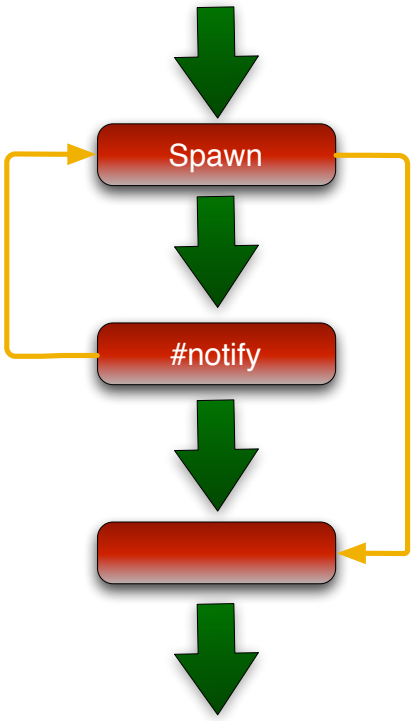
EventMachine spawned processes are inspired by Erlang processes. The naming can be a bit confusing as there are no operating system processes involved. The idea behind spawned processes is that you can create a process and attach some code. At some point in the future you can *#notify* the spawned object and it will execute the block attached.

Unlike deferrables, the code block will not be executed immediately but at some point after the *#notify* call has been made.

```
require 'eventmachine'

EM.run do
  s = EM.spawn do |val|
    puts "Received #{val}"
  end

  EM.add_timer(1) do
    s.notify "hello"
  end
  EM.add_periodic_timer(1) do
    puts "Periodic"
  end
  EM.add_timer(3) do
    EM.stop
  end
end
```




```
Rei:EventMachine dj2$ ruby spawn.rb
Periodic
Received hello
Periodic
```

Network Fun

Now we get into the good stuff. Network programming is what EM has been designed to handle. The ability to handle any protocol, and a series of base protocol implementations, makes things easy as an API or server developer.

Servers

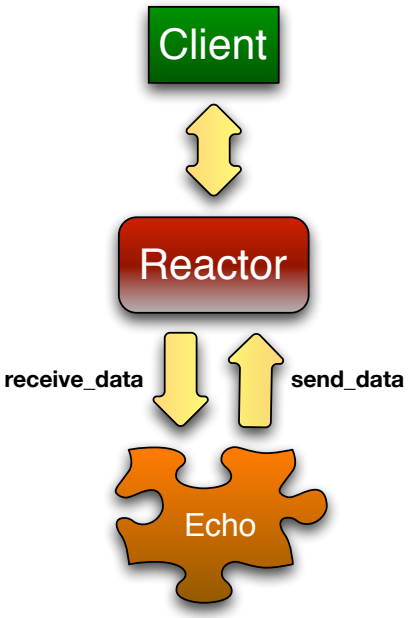
We're going to get started on the server side of things. The code to do clients will be very similar but we'll get to that in the next section.

Heading back to our introduction example of the Echo server:

```
require 'eventmachine'

class Echo < EM::Connection
  def receive_data(data)
    send_data(data)
  end
end

EM.run do
  EM.start_server("0.0.0.0", 10000, Echo)
end
```




```
Rei:~ dj2$ telnet localhost 10000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
helo
helo
goodbye cruel world
goodbye cruel world
```

As you can see, we're creating an *Echo* class and inheriting from *EM::Connection*. We use the *EM#start_server* method to create a server listening on all interfaces, port 10,000 and using the *Echo* class we've defined.

Interestingly, there are actually two more ways we could have written this same code. Which one you use is a matter of taste.

We could use a module:

```
require 'eventmachine'

module Echo
  def receive_data(data)
    send_data(data)
  end
end

EM.run do
  EM.start_server("0.0.0.0", 10000, Echo)
end
```

Or a block:

```
require 'eventmachine'

EM.run do
  EM.start_server("0.0.0.0", 10000) do |srv|
    def srv.receive_data(data)
      send_data(data)
    end
  end
end
```

In all three cases when a new connection is established a new, anonymous, class is created which will include your code. This is important, each connection has a new instance of your class. You can't store anything in that instance between connections as it won't exist on the next connection. We'll get to ways around this in a moment.

We need to implement one method, *#receive_data(data)* in order to function as a server. If you don't implement *#receive_data* you'll end up seeing something similar to "*.....>>>6"* spit out to the console. If you're like me, you'll then spend thirty minutes trying to figure out where that's coming from.

Along with *#receive_data,* there are a few other methods that will be invoked during the lifetime of the client connection.

post_init	Called during instance initialization but before the connection has been fully established.
connection_completed	Called after the connection has been fully established.
receive_data(data)	Called when data is received from the client. Data will be received in chunks. Chunk assembly us you're responsibility.
unbind	Called when the client is fully disconnected.

In our example we use *#send_data(data)* to send the reply back to the client. If we were sending back a large data file we could also use *#send_file_data(filename)* which is a higher performance method to stream large chunks of data.

Finally, although not shown here, two useful methods are *#close_connection* and *#close_connection_after_writing*. These two methods are very similar in their operation. In both cases they will signal for the client connection to be closed. The difference lies in that *#close_connection_after_writing* will make sure that any data sent with *#send_data* is sent to the client before the connection is closed.

As I mentioned previously, we can't store any information in our class object between connections. Luckily, or, I guess, by design, EventMachine provides a mechanism to handle this.

```
require 'eventmachine'

class Pass < EM::Connection
  attr_accessor :a, :b
  def receive_data(data)
    send_data "#{@a} #{data.chomp} #{b}"
  end
end

EM.run do
  EM.start_server("127.0.0.1", 10000, Pass) do |conn|
    conn.a = "Goodbye"
    conn.b = "world"
  end
end
```

```
titania:~ dj2$ telnet localhost 10000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
mostly cruel world
Goodbye mostly cruel world
```

By providing a block to *EM#start_server,* EventMachine will pass in the *Pass* instance after it has been initialized but before any client data has been received. We can use this object to set any state into our instance that we require.

Clients

Once we've got our server up it would be beneficial to connect to it with a client. Thankfully, you know the majority of what's needed to make the magic happen after doing our server work.

```
require 'eventmachine'

class Connector < EM::Connection
  def post_init
    puts "Getting /"
    send_data "GET / HTTP/1.1\r\nHost: MagicBob\r\n\r\n"
  end

  def receive_data(data)
    puts "Received #{data.length} bytes"
  end
end

EM.run do
  EM.connect("www.postrank.com", 80, Connector)
end
```

```
titania:EventMachine dj2$ ruby connect.rb
Getting /
Received 1448 bytes
Received 1448 bytes
Received 1448 bytes
Received 1448 bytes
Received 2896 bytes
Received 1448 bytes
Received 1448 bytes
Received 935 bytes
```

Other then the call to *EM#connect* to create the connection, the code in this example follows the ideas set forth when creating our server above. The callback methods have the same names and you send data to the server the same as you would to a client, with *#send_data*.

Conclusion

This ends our whistle-stop tour of EventMachine. We've taken a look at getting things up and running using *EM#run*. Creating one-shot and periodic timers. Deferring and next_ticking blocks of code. Creating deferrables and spawned processes along with client and server code.

Hopefully, after all that, you have a better understanding of how EventMachine function and how it can be used in your applications.

If you've got any questions or comments feel free to contact me at: dan@aiderss.com.

Resources

- http://rubyeventmachine.com/
- http://github.com/eventmachine/eventmachine/tree/master
- http://groups.google.com/group/eventmachine/topics
- http://eventmachine.rubyforge.org/
- http://www.igvita.com/2008/05/27/ruby-eventmachine-the-speed-demon/
- http://20bits.com/articles/an-eventmachine-tutorial/
- http://nutrun.com/weblog/distributed-programming-with-jabber-and-eventmachine/
- http://www.infoq.com/news/2008/06/eventmachine
- http://everburning.com/news/playing-with-eventmachine/
- http://blog.nominet.org.uk/tech/2007/10/12/dnsruby-and-eventmachine/
- http://devver.net/blog/2008/10/sending-files-with-eventmachine/
- http://en.oreilly.com/rails2008/public/schedule/detail/1820
- http://nhw.pl/wp/2007/12/07/eventmachine-how-to-get-clients-ip-address
- http://simonwex.com/articles/2008/10/22/eventmachine-http-client
- http://adrianhosey.blogspot.com/2009/01/eventmachine-tcp-server-module-for-ruby.html