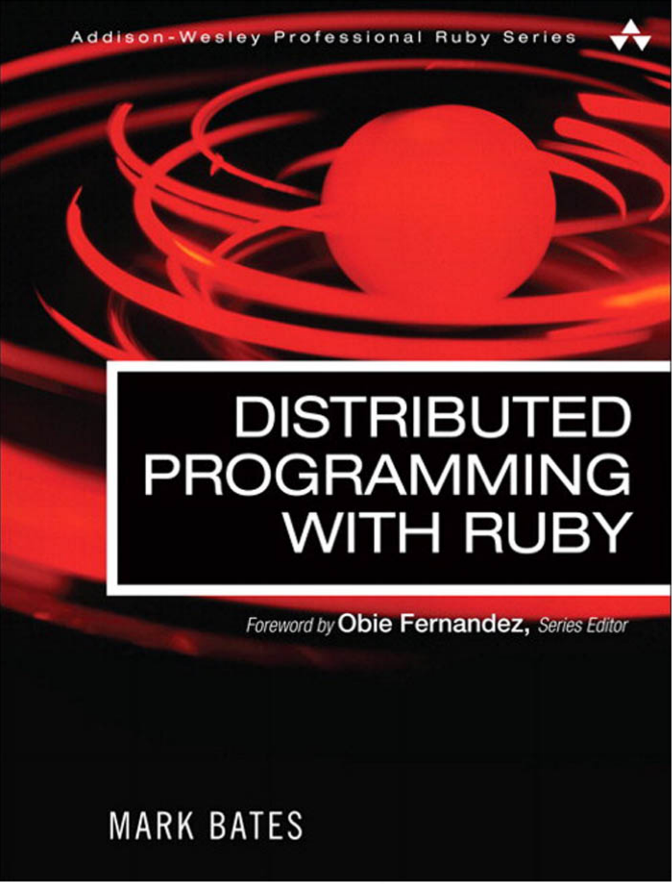
Addison-Wesley Professional Ruby Series

DISTRIBUTED PROGRAMMING WITH RUBY

Foreword by Obie Fernandez, Series Editor

MARK BATES



Distributed Programming with Ruby

Mark Bates

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data:

Bates, Mark, 1976-
Distributed programming with Ruby / Mark Bates.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-63836-6 (pbk. : alk. paper) 1. Ruby (Computer program language) 2. Electronic data processing—Distributed processing. 3. Object-oriented methods (Computer science) I. Title.
QA76.73.R83B38 2010
005.1'17—dc22

2009034095

ISBN-13: 978-0-321-63836-6
ISBN-10: 0-321-63836-0
Text printed in the United States on recycled paper at RR Donnelley and Sons in Crawfordsville, Indiana
First printing November 2009

Editor-in-Chief
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Senior Project Editor
Lori Lyons

Copy Editor
Gayle Johnson

Indexer
Brad Herriman

Proofreader
Apostrophe Editing Services

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

To Rachel, Dylan, and Leo.
Thanks for letting Daddy hide away until the wee hours of the morning
and be absent most weekends. I love you all so very much,
and I couldn't have done this without the three of you.

This page intentionally left blank

Contents

Foreword

Mark's career in programming parallels mine to a certain degree. We both started developing web applications in 1996 and both did hard time in the Java world before discovering Ruby and Rails in 2005, and never looking back.

At RubyConf 2008 in Orlando, I toasted Mark on his successful talk as we sipped Piña Coladas and enjoyed the "fourth track" of that conference—the lazy river and hot tub. The topic of our conversation? Adding a title to the Professional Ruby Series in which Mark would draw from his experience building Mack, a distributed web framework, as well as his long career doing distributed programming. But most important, he would let his enthusiasm for the potentially dry subject draw in the reader while being educational. I sensed a winner, but not only as far at finding the right author. The timing was right, too.

Rails developers around the world are progressing steadily beyond basic web programming as they take on large, complex systems that traditionally would be done on Java or Microsoft platforms. As a system grows in scale and complexity, one of the first things you need to do is to break it into smaller, manageable chunks. Hence all the interest in web services. Your initial effort might involve cron jobs and batch processing. Or you might implement some sort of distributed job framework, before finally going with a full-blown messaging solution.

Of course, you don't want to reinvent anything you don't need to, but Ruby's distributed programming landscape can be confusing. In the foreground is Ruby's DRb technology, part of the standard library and relatively straightforward to use—especially for those of us familiar with parallel technologies in other languages, such as Java's RMI. But does that approach scale? And is it reliable? If DRb is not suitable for your production use, what is? If we cast our view further along the landscape, we might ask: "What about newer technologies like AMQP and Rabbit MQ? And how do we tie it all together with Rails in ways that make sense?"

Mark answers all those questions in this book. He starts with some of the deepest documentation on DRb and Rinda that anyone has ever read. He then follows with coverage of the various Ruby libraries that depend on those building blocks, always keeping in mind the practical applications of all of them. He covers assembling cloud-based servers to handle background processing, one of today's hottest topics in systems architecture. Finally, he covers the Rails-specific libraries BackgrounDRb and Delayed Job and teaches you when and how to use each.

Ultimately, one of my most pleasant surprises and one of the reasons that I think Mark is an up-and-coming superstar of the Ruby community is the hard work, productivity, and fastidiousness that he demonstrated while writing this book. Over the course of the spring and summer of this year, Mark delivered chapters and revisions week after week with clockwork regularity. All with the utmost attention to detail and quality. All packed with knowledge. And most important, all packed with strong doses of his winning personality. It is my honor to present to you the latest addition to our series, *Distributed Programming with Ruby*.

Obie Fernandez, Series Editor
September 30, 2009

Preface

I first found a need for distributed programming back in 2001. I was looking for a way to increase the performance of an application I was working on. The project was a web-based email client, and I was struggling with a few performance issues. I wanted to keep the email engine separate from the client front end. That way, I could have a beefier box handle all the processing of the incoming email and have a farm of smaller application servers handling the front end of it. That seems pretty easy and straightforward, doesn't it? Well, the language I was using at the time was Java, and the distributed interface was RMI (remote method invocation). Easy and straightforward are not words I would use to describe my experiences with RMI.

Years later I was working on a completely different project, but I had a not-too-dissimilar problem—performance. The application this time was a large user-generated content site built using Ruby on Rails. When a user wrote, edited, or deleted an article for the site, it needed to be indexed by our search engine, our site map needed to be rebuilt, and the article needed to be injected into the top of our rating engine system. As you can imagine, none of this was quick and simple. You can also probably guess that our CEO wanted all of this to happen as close to real time as possible, but without the end user's having to wait for everything to get done. To further complicate matters, we had limited system resources and millions of articles that needed to be processed.

I didn't want to burden our already-overworked applications server boxes with these tasks, so I had to offload the processing to another machine. The question came to be how I could best offload this work. The first idea was to use the database as the transfer mechanism. I could store all the information in the database that these systems would need. Then the machine that was to do the processing could poll the database at a regular interval, find any pending tasks, pull them out of the database, create the same heavy objects I already had, and then start processing them. The problem, as you most likely already know, is that I'm now placing more load on the database. I would be polling it continually, regardless of whether it contained any tasks. If it did have tasks, I would have to pull those records out of the database and use more system resources transforming the records back into those same heavy Ruby objects I already had.

What I really wanted to do was just send the fully formed Ruby objects I had already created to the other machine and let it do the processing. This would lessen the burden all around. In addition to the lighter load on the database, memory, and system resources, the machine doing the processing would work only when it was told to, and it wouldn't waste recourses by continually polling the database. Plus, without polling, the parts of the application the CEO wanted updated in near real time would get updated faster.

After I realized that what I wanted to do was to use some sort of distributed mechanism, that's when I decided to see what sort of RMI-esque features Ruby had. I was already impressed with Ruby for being a terse language, but when I found the `DRb` (Distributed Ruby, also known as `dRuby`) package, I became a believer. I found that writing distributed applications in Ruby could be simple, and dare I say fun.

Who Is This Book For?

This book is quite simply written for the intermediate to advanced Ruby developer who wants to start developing distributed applications. This book assumes that you have good knowledge of Ruby, at least at the intermediate developer level. Although we will touch on some parts of the Ruby language—particularly those that might be confusing when dealing with distributed applications—we will not be going into the language in depth.

Although you should know Ruby, this book assumes that you probably do not understand distributed programming and that this is your first venture into this world. If you have done distributed programming before, this book will help you quickly understand how to do it in Ruby. If you haven't, this book will help you understand what distributed programming is and isn't.

How Is This Book Organized?

This book is split into four parts. Part I examines what ships with the standard library in Ruby 1.8.x and beyond. We look, in depth, at understanding how DRb (dRuby or Distributed Ruby) and Rinda work. We will build some simple applications in a variety of ways and use those examples to talk about the libraries. We examine the pros and cons of DRb and Rinda. By the end of Part I, "Standard Library," you should feel comfortable and ready to build your distributed applications using these libraries.

Part II, "Third-Party Frameworks and Libraries," looks at a variety of third-party tools, libraries, and frameworks designed to make distributed programming in Ruby easy, fun, and robust. Some of these libraries build on the DRb and Rinda libraries we learned about in Part I, and others don't. Some are based on executing arbitrary code on another machine. Others are based on running code in the background to elevate performance.

Part III, "Distributed Message Queues," takes a close look at some of the leading distributed message queues available to the Ruby community. These queues can help facilitate communication and tasks between your applications. Distributed message queues can help increase your applications' performance by queuing up work to be done at a later date instead of at runtime.

Finally, Part IV, "Distributed Programming with Ruby on Rails," looks at a few libraries that are designed to work exclusively with the Ruby on Rails web framework. These libraries might already be familiar to you if you have been using Ruby on Rails for several years. But there is always something to be learned, and that's what the chapters in this part of this book will help you with.

During the course of the book, we will examine a breadth of different technologies; however, this book is not necessarily a how-to guide. Instead, you will use these different technologies to help understand the complex problems associated with distributed programming and several different ways you can solve these problems. You'll use these technologies to learn about RMI, message queues, and MapReduce, among others.

How to Run the Examples

I have tried to make this book as easy to use and follow as possible. When a new technology is referenced or introduced, I give you a link to find out more about it and/or its developer(s). When you see a code sample, unless otherwise stated, I present that sample in its entirety. I have also taken extra effort to make sure that you can easily run each of those code samples as is. Unless otherwise stated, you should be able to take any code sample, copy it into a Ruby file, and run it using the `ruby` command, like this:

```
$ ruby foo.rb
```

There are times when a file needs to be named something specific or has to be run with a special command. In particular, Chapter 4, "Starfish," covers this issue. At that time I will call your attention to these details so that you can run the examples without hassle.

In some chapters, such as Chapters 2, "Rinda," and 8, "AMQP/RabbitMQ," background servers need to be run for the examples to run correctly. It is highly recommended that you restart these background servers between each set of examples that are presented in these chapters. A lot of these chapters iteratively build on a piece of software, and restarting the servers between runs helps eliminate potentially confusing results.

Acknowledgments

Writing a book isn't easy. I know that's an obvious statement, but sometimes I think people just don't quite get what goes into writing a book. I didn't think it would be this difficult. Thankfully, though, I have somehow made it out the other side. I'm a little (more like a lot) battered, bruised, and very tired, but it was definitely worth it.

However, I couldn't have done this without a lot of help from a lot of different people. As with a good Oscar speech, I'll try to keep this brief, and I'm sure, as with an Oscar speech, I'll leave out some people. If I've left you out, I apologize. Now, let's see if I can get through this before the orchestra plays me off.

First, and foremost, I have to thank my family. Rachel, my beautiful wife, has been so supportive and understanding, not just with this book, but with everything I do. I know that she would've loved to have had me spend my weekend afternoons going for walks with her. Or to have me do the stuff around the house that needs to get done. Instead, she let me hide away in my office/studio, diligently (sometimes) working on my book. The same goes for Dylan, my son. I'm sure he would've preferred to have Daddy playing with him all day. I'm all yours now, little buddy. And to little Leo: This book and you share a very similar timeline—only two days separate your birth and this book going to print. Welcome, son! Your mother and big brother will tell you this hasn't been easy, and you're better for having slept through the whole thing.

Before I get off the subject of family, I would like to thank my parents. The reasons are obvious. They brought me into this world. (And, from what I've been told, they can take me out as well.) They have always supported me and have made me the man I am today. Because of them I am not afraid to take risks. I'm not afraid to fail. In general, I'm not afraid. Except for dogs. I'm afraid of dogs, but I don't think that's my parents' fault.

I would also like to quickly thank the rest of my friends, family, and coworkers. Mostly I'm thanking them for not telling me to shut up whenever I started talking about my book, which, let me tell you, was a lot. Even I got tired of hearing about it!

In November 2008, I gave a presentation titled "Building Distributed Applications" at RubyConf in Florida. After my presentation I was approached by a couple of gentlemen telling me how much they enjoyed my talk. They wanted to know where they could find out more about `DRb` and `Rinda`. I told them that unfortunately very little documentation on the subject existed—just a few blog posts here and there, and the code itself. They told me I should write a book about distributed programming with Ruby, adding that they would order it in a heartbeat. I thought it was a great idea. Shortly before I sent my manuscript to the publisher, I received an email from one of these gentlemen, Ali Rizvi. He had stumbled across one of my blog posts on a completely unrelated subject (the iPhone), and he realized who I was and that I was writing this book. He dropped me a quick note to say hi and that he was looking forward to reading the book. So Ali, now that I know your name, thank you for the idea!

At that same conference I found myself having a few drinks in the hot tub with none other than Obie Fernandez, the Professional Ruby Series editor for Addison-Wesley. He told me how much he enjoyed my presentation earlier that day. I used the opportunity to pitch him my book idea—the one I'd had only an hour before. He loved the idea and told me he thought it would be a great book, and he would love to be a part of it. A few weeks later I received an email from Debra Williams Cauley at Addison-Wesley, wanting to talk to me about the book. The rest, as they say, is history.

Obie and Debra have been my guiding light with this book. Obie has given me great advice and guidance on writing it. His direction as a series editor has been invaluable. Thank you, Obie, for your mentoring, and thank you for helping me get this opportunity.

Debra, thank you. Thank you so much. Debra managed this book. She answered all my questions (some good, some bad); she was always there with an answer. She never told me a request was too outrageous. She helped guide me through the treacherous waters of book writing, and it's because of her that I managed to make it through to the other end mostly unscathed. I can't say enough great things about Debra, and I know I can never thank her as much as she deserves to be thanked in regards to this book. Thank you, Debra.

I would like to thank Songlin Qiu. Songlin's amazing technical editing is, quite frankly, what made this book readable. She constantly kept me on my toes and made sure not only that the book was consistent, but also that it was well written and worth reading. I'm pretty sure she also fixed a million misuses of the "its" that appeared in the book. Thank you, Songlin.

Gayle Johnson also deserves a thank you here for her copy editing. She is the one who turned my words into poetry. Well, maybe poetry is an exaggeration, but trust me—this book is a lot more enjoyable to read because of her. She turned my Guinness soaked ramblings into coherent English. Thank you, Gayle.

Lori was my project editor on this book. She helped to guide me through the murky waters that are the copy editing/pre-production phase of writing a book. Thank you, Lori, for helping me take my book to the printer.

I would like to acknowledge another group of people—technical reviewers. They read the book and told me all the things they don't like about it. Just kidding—sort of. They are my peers. Their job is to read the book and give me feedback on what they liked, disliked, and were indifferent to. Their comments ranged from "Why didn't you talk about such-and-such?" to "I like how you flow from this subject to that one." Some of these people I came to absolutely love, either because they offered me great advice or because they liked what I had done. Others I came to be frustrated with, either because I didn't like their comments or because they were right, and I don't like being wrong. Either way, all the feedback was extremely helpful. So with that said, here is a list of those people, in no particular order: Gregg Pollack, Robert P.J. Day, Jennifer Lindner, and Ilya Grigorik. Thank you all so very much.

I want to thank everyone at Addison-Wesley who worked on this book. Thank you to those who dedicated their time to making my dream into a reality. I know there are people who are working hard in the background that I am unaware of, from the cover art, to the technical editing, to the page layout, to the technical reviewers, to the person who corrects my spelling, thank you.

Finally, thank *you*. Thank you for spending your time and your money on this book. I appreciate it very, very much.

About the Author

Mark Bates has been developing web applications of one kind or another since 1996. He has spent an ungodly amount of time programming Java, but thankfully he discovered Ruby in late 2005, and life has been much nicer since.

Since discovering Ruby, Mark has become a prominent member of the community. He has developed various open-source projects, such as Configatron, Cachetastic, Genosaurus, APN on Rails, and the Mack Framework, just to name a few. The Mack Framework brought Mark to the forefront of distributed programming in the Ruby community. Mack was a web framework designed from the ground up to aid in the development of distributed applications.

Mark has taught classes on both Ruby and Ruby on Rails. He has spoken at several Ruby gatherings, including 2008's RubyConf, where he spoke about building distributed applications.

Mark has an honors degree in music from the Liverpool Institute for Performing Arts. He still likes to rock out on the weekends, but set times are now 10 p.m., not 2 a.m.

He lives just outside of Boston with his wife Rachel and their sons, Dylan and Leo, both of whom he missed very much when writing this book.

Mark can be found at http://www.markbates.com and http://github.com/markbates.

This page intentionally left blank

CHAPTER 5
Distribunaut

In early 2008, I was working for a company that was using Ruby on Rails as the framework behind the application we were building. For the most part we were happy with Rails, but there were things we wanted to do that Rails was just not a good fit for. First we realized that what had started as a Web 2.0 application was anything but that. Instead, we came to the conclusion that we were building a rather large portal application.

For all of its pros, Rails has a few cons as well. I won't go into all of them now, but the biggest disadvantage we found was that Rails doesn't want to help you write complex portal applications. It wants you to build smaller, simpler applications—at least, at the time it did. With Rails 3.0 on the horizon, that may change.

In addition to building this large portal, we decided we wanted to split our application into many applications. The advantages we saw were smaller code bases that were easier to maintain and separate applications that were easier to scale. We also could push updates and new features sooner, because we didn't have a gigantic code base to worry about.

We identified three main problems. First, we wanted to let each application maintain its own set of routing, but we wanted the other applications to be able to use the dynamic routing we had become accustomed to in Rails. We didn't want to hardcode URLs in the other applications; we wanted them generated by the application they would be linking to. Second, we wanted to share views and layouts among these applications. We didn't want to have to deal with SVN externals, GIT submodules, or symlinks. We wanted to be able to quickly say, "Here is a URL for a layout. Render it like you would a local layout." Finally, we wanted to share models and libraries throughout all these applications without having to worry about packaging them and redeploying all these applications each time we made a bug fix to a model.

With these goals in mind, I set out to find a Ruby web framework that would help us achieve these goals. After downloading and testing nearly 20 frameworks, I was at a loss for the solution we needed. Then I found Rack.[1] Rack bills itself as a framework for frameworks. It is a middleware abstraction layer that lets framework developers get on with developing their framework without worrying about things like parsing requests and talking with application servers. Within a few hours I had a simple MVC-based framework up and running, and the Mack Framework was born.

I then spent the next year building and developing a large feature set for Mack, including all the libraries to handle distributed routes, views, and models. During that time I was asked time and again to make these components available outside the Mack framework for others to use. In April 2009, I announced an early version of a library I dubbed Distribunaut.

Distribunaut[2] is a port of one-third of the distributed features that are found in Mack. In particular, it focuses on making it incredibly easy to distribute models and other Ruby classes. You will not find distributed views/layouts and routes in Distribunaut. The reason is that they are too specific to each of the web frameworks out there, and coding for each one would be a lot of work.

So with that brief history of Distribunaut, let's look at what it can do for us.

Installation

Installing the Distribunaut library is simple. It can be installed using RubyGems:

```
$ gem install markbates-distribunaut -s http://gems.github.com
```

You should then see something similar to the following, telling you that you have successfully installed the gem:

```
Successfully installed markbates-distribunaut-0.2.1
```

Blastoff: Hello, World!

Distribunaut uses DRb and Rinda to do most of its heavy lifting. The good news is that because you have already learned all about DRb and Rinda, you can easily jump into experimenting with Distribunaut.

As you'll remember from our look at Rinda, we need to start a RingServer before we can run any code. Distribunaut ships with a convenient binary to help make starting, stopping, and restarting a RingServer easy:

```
$ distribunaut_ring_server start
```

If you wanted to stop the RingServer, you would do so with the following command:

```
$ distribunaut_ring_server stop
```

You can probably guess how to restart the server. You should restart the RingServer between all these examples, just so things don't go a bit funny on you:

```
$ distribunaut_ring_server restart
```

So, with a RingServer running nicely as a daemon in the background, let's kick off things with a simple "Hello World" application. Let's start with a server. Keep in mind that, as we talked about earlier in the book, when we are using DRb and Rinda, applications can act as both a server and a client. So when we use the term "server" here, we are merely using it to describe a bit of code that serves up some content. So what does our HelloWorld class look like with Distribunaut? Let's see:

```
require 'rubygems'
require 'distribunaut'

configatron.distribunaut.app_name = :hello_world_app

class HelloWorld
  include Distribunaut::Distributable
  def say_hi
    "Hello, World!"
  end

end

DRb.thread.join
```

First we require `rubygems` and then the `Distribunaut` library itself. After that we hit the first of two lines that make Distribunaut special.

Each Distribunaut "application" needs a unique name. When we talk about applications within Distribunaut, we are actually talking about a Ruby VM/process that contains one or more Distribunaut classes. The name of that application should be unique to avoid confusion. We will look at what can happen with redundant application, and class, names a bit later in this chapter.

To manage its configurations, Distribunaut uses the Configatron[3] library. We set the application as follows:

```
configatron.distribunaut.app_name = :hello_world_app
```

This needs to happen only once per Ruby VM. If you set it multiple times, strange things can happen, so be careful. In our sample code we are setting the application name to `:hello_world_app`. We could just as easily set it to something like `:server1` if we wanted to make it more generic for other Distribunaut classes we were planning on running in the same Ruby VM.

After we have set up our application name, the only other thing we have to do is include the `Distribunaut::Distributable` module in our `HelloWorld` class. Then we are ready to try to get a "Hello, World!" remotely.

Before we get to our client code, let's take a quick look at what the preceding `HelloWorld` class would've looked like had we used raw `DRb` and `Rinda`:

```
require 'rinda/ring'

class HelloWorld
  include DRbUndumped

  def say_hi
    "Hello, World!"
  end
```

```
end

DRb.start_service
ring_server = Rinda::RingFinger.primary
ring_server.write([:hello_world_service, :HelloWorld,
                   HelloWorld.new, 'I like to say hi!'],
                  Rinda::SimpleRenewer.new)

DRb.thread.join
```

Although the `HelloWorld` class part of it is relatively the same, much more noise is required at the end to get our `HelloWorld` instance into the `RingServer`. At this point it is also worth pointing out that in the `Rinda` version of `HelloWorld` we had to create a new instance of the class. This means that we can't call any class methods that `HelloWorld` may have. This includes the ability to call the `new` method and get a new instance of the `HelloWorld` class. We are stuck with that instance only. We did not do anything of the sort with the Distribunaut version of the class. In fact, you probably have noticed that we didn't make any calls to get it into the `RingServer`. We'll talk about why that is shortly. First, let's look at our client code:

```
require 'rubygems'
require 'distribunaut'

hw = Distribunaut::Distributed::HelloWorld.new
puts hw.say_hi
```

If we were to run this code, we should see the following output:

```
Hello, World!
```

What just happened there? Where did the `Distribunaut::Distributed::HelloWorld` class come from? How did it know to print "Hello, World!" when we called the `say_hi` method? All great questions.

The `Distribunaut::Distributed` module is "special." When you preface a constant such as `HelloWorld` in that module, it queries the `RingServer` and attempts to find a service that matches that constant. So, in our case it searched the `RingServer` for a service called `HelloWorld`. It found the `HelloWorld` class we created earlier and returned a reference to it. With that reference we could call the `new` method on that class, which returned a new instance of the HelloWorld class. And then we could call the say_hi method.

So if we didn't explicitly place our HelloWorld class in the RingServer, how did we access it? And how were we able to call a class method on it, when we know that you have to put instances only into a RingServer? The same answer applies to both questions. When we included the Distribunaut::Distributable module into the HelloWorld class, it created a Singleton wrapper class on-the-fly that then proxies all methods called on that proxy class onto the original HelloWorld class. With that we can put the Singleton instance into the RingServer. Then we can call class methods, which allows us to do things like call the new and get back a new instance of the class.

Having all of this happen automatically also helps clean up the usual supporting code you need to write to both set an instance into the RingServer and retrieve that instance later. Just look at what a plain-vanilla DRb and Rinda implementation of the client would look like:

```
require 'rinda/ring'

DRb.start_service
ring_server = Rinda::RingFinger.primary

service = ring_server.read([:hello_world_service,
                           nil, nil, nil])
server = service[2]

puts server.say_hi
```

This is not only more code, but also uglier code.

Building a Distributed Logger with Distribunaut

So now that you have a good understanding of how Distribunaut works, and what it does under the covers, let's try to create a distributed logger and see how it goes. To create our distributed logger, we want to create a RemoteLogger class. Here's what that would look like:

```
require 'rubygems'
require 'distribunaut'
require 'logger'
```

```
configatron.distribunaut.app_name = :remote_logger

LOGGER = ::Logger.new(STDOUT)

class RemoteLogger
  include Distribunaut::Distributable

  class << self

    def new
      LOGGER
    end

    [:debug, :info, :warn, :error, :fatal].each do |meth|
      define_method(meth) do |*args|
        LOGGER.send(meth, *args)
      end
    end

  end

end

DRb.thread.join
```

Although this looks a lot more intimidating than our `HelloWorld` class, it really isn't. The extra code comes from making it a bit easier to access the underlying Ruby `Logger` class we want to wrap. We could have just harnessed the incredible power of Ruby and opened up the `Logger` class and included the `Distribunaut::Distributable` module directly into it, but that is generally not considered good practice. Besides, this way lets us talk about a few things we couldn't talk about otherwise. Let's look at it in a bit more depth; you'll see it isn't that complex.

After we require the correct classes and define our application name (this time we are calling it `:remote_logger`), we create a constant called `LOGGER` to act as a holder for our `Logger` instance. We want only one instance of the `Logger` class. That is why we assign it to the global constant—so that we can access it throughout the rest of our code.

After we have included the `Distribunaut::Distributable` module into our `RemoteLogger` class, we then add a few methods for convenience. The first of these methods is a class-level override of the `new` method. We do this so that when our clients try to create a new instance of the `RemoteLogger` class, they are actually getting the wrapped `Logger` class instead. Next we generate the five standard logging methods on `Logger`, putting them at the class level of the `RemoteLogger` class. These methods simply proxy the methods onto the single instance of our `Logger` class that we have stored in our `LOGGER` constant. We do this so that our clients can call these methods at the class level of `RemoteLogger` without having to create a new instance of it. This is easier to demonstrate in the client code.

With all of that out of the way, let's see what our client code would look like:

```
require 'rubygems'
require 'distribunaut'

logger = Distribunaut::Distributed::RemoteLogger.new

logger.debug("Hello, World!")

Distribunaut::Distributed::RemoteLogger.error("oops!")
```

In this client we first create a new "instance" of the `RemoteLogger` class. I put "instance" in quotes for a reason. Remember that we don't actually get a new instance of the `RemoteLogger` class. Instead, we simply get back a reference to the global instance of the `Logger` class we set up earlier.

As soon as we have the `RemoteLogger`, we can call the standard logging methods, such as `debug`. We should see our message printed to the server's screen, not the client's. After we call the `debug` method, we call the class method `error` on the `RemoteLogger` class and pass it the message "oops!".

If we were to run all of this, we would get the following:

```
Hello, World!
oops!
```

As you can see, creating a new distributed logger with Distribunaut is actually quite easy. We could have simplified the code by not giving class-level convenience methods for the common logging methods. But it was only a few more lines of code, and it could make the end user's life a little easier.

Avoiding Confusion of Services

Earlier, when speaking about application names, I mentioned that names need to be unique to avoid confusion, but I didn't explain what I meant.

You know from Chapter 2, "Rinda," that when we create a `Tuple` to put into the `RingServer`, we give it some unique characteristics that allow us to retrieve it easily. The combination of these characteristics becomes sort of like an ID for that particular `Tuple`. So imagine if we were to put two `Tuples` into the `RingServer` that had the same characteristics. How would we retrieve the specific one we want? If we use the same application name, we not only run the risk of overwriting another `Tuple`, but we also make it difficult to find later.

As you have seen, Distribunaut performs a lot of magic that keeps us from having to write as much code. It also makes the code we write cleaner and easier to use and maintain. One thing Distribunaut does for you is build the search characteristics for you when you make a call to the special `Distribunaut::Distributed` module. When Distribunaut goes to build the search parameters for that request, it takes into account only the class, or service, name you provide. Because of this, if you have two applications serving up a class with the same name, you are unsure which one you will receive from the query. In some cases this might be fine, but in other cases, it might be a problem.

Let's look at a simple example. Let's create a service that serves up a `User` class. We want to launch at least two instances of this service for this example. To do that we need to run the following code twice to start two instances of it:

```
require 'rubygems'
require 'distribunaut'

user_servers = ['0']

services = Distribunaut::Utils::Rinda.available_services

services.each do |service|
  if service.app_name.to_s.match(/^user_server_(\d+)$/)
    user_servers << "#{$1}"
  end
end
```

```
user_servers.sort!

app_name = "user_server_#{user_servers.last.succ}"

puts "Launching: #{app_name}"

configatron.distribunaut.app_name = app_name

class User
  include Distribunaut::Distributable

  def app_server_name
    configatron.distribunaut.app_name
  end

end

DRb.thread.join
```

A large majority of the preceding code simply finds out what the last service, if there is one, was called. Then it names the service that is currently launching so that it has a unique name. Although most of this is straightforward Ruby code, it is worth pointing out the call to the `available_services` method on the `Distribunaut::Utils::Rinda` module. The `available_services` method, as its name implies, returns an `Array` of the services that are currently registered with Distribunaut. The elements of this `Array` are `Distribunaut::Tuple` classes, which are simply a convenience class to make it easier to deal with `Tuples` that are in the Distribunaut format.

After we have decided on an appropriate application name and registered it, we create a `User` class, include the `Distribunaut::Distributable` module and give it an instance method that returns the application name that is running this service.

Now, with a couple of instances of our service running, let's look at the style of client we have been using so far in this chapter:

```
require 'rubygems'
require 'distribunaut'

user = Distribunaut::Distributed::User.new
puts user.app_server_name
```

So which instance of the user service do we get when we run this? Well, on my system I see the following printed:

```
user_server_1
```

On your system you might see this:

```
user_server_2
```

or another variation, depending on how many instances you have running. There is no guarantee which instance you will receive when accessing services this way. Again, this might be acceptable in your environment, or it might not.

So what do you do when this is unacceptable, or you want to get a specific instance of a service? Distribunaut provides you with a method called `lookup`. This method is found on the `Distribunaut::Distributed` module. The `lookup` method takes a URL to find the specific instance you are looking for.

Right about now you should be wondering how you are supposed to know the URL of the service you want to look up. Don't worry. Distribunaut has you covered by making it easy to look up these services. Let's look at a client that wants to find specific instances of the user services we have running:

```
require 'rubygems'
require 'distribunaut'

user_one_url = "distributed://user_server_1/User"
UserOne = Distribunaut::Distributed.lookup(user_one_url)

user_two_url = "distributed://user_server_2/User"
UserTwo = Distribunaut::Distributed.lookup(user_two_url)

user1 = UserOne.new
puts user1.app_server_name

user2 = UserTwo.new
puts user2.app_server_name
```

Building the URL for the service we want is quite simple. The format is `distributed://<application_name>/<service_name>`. Because of this format, it is important that we have unique application names for each Ruby VM so that we can easily seek out the one we are looking for.

With the URLs in hand for the two services we are looking for, we can call the `lookup` method and find these two services. When we have them, we can create new instances of the `User` class and print the return value of the `app_server_name` method. You should see something similar to the following printed:

```
user_server_1
user_server_2
```

With the `lookup` method now in our arsenal, we can code with confidence, knowing that we will always get the specific instance of a service we are looking for. And we can do it without having to deal with IP addresses, ports, and other such nonsense.

Borrowing a Service with Distribunaut

As you probably remember from Chapter 2, when we retrieve `Tuples` from the `RingServer`, we have two choices. We can either read the `Tuple` or take the `Tuple`. The former leaves the `Tuple` in the `RingServer` for others to access simultaneously. The latter removes the `Tuple` from the `RingServer`; as a consequence, no one else can gain access to that `Tuple`.

So what happens when we access a service using Distribunaut? Are we doing a read or a take from the `RingServer`? Distribunaut does a read from the `RingServer`, thereby allowing others to access the same service at the same time.

Most of the time this is the exact behavior you want. You usually want to be a good citizen and let others access the service you are accessing as well. Sometimes, however, you might need to grab hold of a service exclusively, do a few things with that service, and then return it to the `RingServer` for others to have access to.

So how do we take a service from the `RingServer`, use that service, and then return it for wider use? We could use raw `Rinda` and `DRb` code, but that would be ugly, and prone to error should any of the underpinnings of Distribunaut change. Instead, Distribunaut offers the concept of borrowing a service.

To demonstrate how to borrow a service, let's use our simple `HelloWorld` class as the service we want to borrow:

```
require 'rubygems'
require 'distribunaut'

configatron.distribunaut.app_name = :hello_world_app

class HelloWorld
  include Distribunaut::Distributable

  def say_hi
    "Hello, World!"
  end

end

DRb.thread.join
```

Here is what our client code would look like to borrow the `HelloWorld` service:

```
require 'rubygems'
require 'distribunaut'

Distribunaut::Distributed::HelloWorld.borrow do |hw_class|
  hw = hw_class.new
  puts hw.say_hi
  begin
    hw = Distribunaut::Distributed::HelloWorld.new
  rescue Rinda::RequestExpiredError => e
    puts "Oops - We couldn't find the HelloWorld class!"
  end

end

hw = Distribunaut::Distributed::HelloWorld.new
puts hw.say_hi
```

If we were to run this code, we should see the following printed:

```
Hello, World!
Oops - We couldn't find the HelloWorld class!
Hello, World!
```

So exactly what did we do, and how did it work? The `borrow` method takes a block and yields a reference to our proxy service, as we discussed earlier in this chapter. This works the same way as if we had called `Distribunaut::Distributed::HelloWorld` directly. The difference is, before the block gets executed, the service is located and removed from the `RingServer`. It is then "locked" and placed back into the `RingServer` in such a way that others can't retrieve it. After the block finishes executing, the service is unlocked in the `RingServer` and is available again for public consumption.

If we look at what is happening in the block, we see that we call the `new` method on the `hw_class` variable, which is the reference to the `HelloWorld` service. The `new` method creates a new instance of the `HelloWorld` class, and we can call the `say_hi` method on it.

To demonstrate that we can't access the `HelloWorld` service directly, we attempt to call it, but, as we see, it raises a `Rinda::RequestExpiredError` because the service cannot be found.

After the block has finished executing, we again try to access the `HelloWorld` service as we would normally. This time, instead of an exception, we are greeted with a pleasant "Hello, World!".

As you can see, the concept of borrowing a service allows us to easily take control of the service we want, do the work we need to do on that service, and then have it automatically returned to the `RingServer` for others to use. It also has the added benefit of being quite easy to code. We don't have to write fragile code that takes the service from the `RingServer`, handles exceptions that may arise, and ensures that the service gets placed back into the `RingServer` correctly.

Conclusion

Obviously I'm slightly biased in my feelings about Distribunaut, seeing as how I am the developer of the library. With that said, I feel strongly that Distribunaut makes distributed objects incredibly easy to code, use, and maintain.

The library continues to grow and develop. Its fundamentals were pulled from the mack-distributed gem for the Mack Framework, but the library has grown and evolved much since its origins. Even during the course of writing this chapter, I found several bugs, enhancements, and performance improvements that could be made, so I made changes. The underpinnings of this library have been working hard in several production environments and have proven themselves to be reliable, fast, and easy to use.

Overall I feel that the simple interface, basically just including a module, makes an already easy system for building distributed applications, `DRb` and `Rinda`, even easier. Instead of having to write code to look up services, read them, parse them, manage them, and so on, you can use something you are already familiar with—simple Ruby objects.

What does the future hold for Distribunaut? As far as the feature set is concerned, that is hard to say. I try to develop features that will actually be used, not features that I think are cool. What I can tell you for sure is that Distribunaut will continue to be maintained and grown to keep up with the challenges of developing distributed applications.

Endnotes

1. http://rack.rubyforge.org/
2. http://github.com/markbates/distribunaut/tree/master
3. http://github.com/markbates/configatron/tree/master

This page intentionally left blank

Index