Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

16. Web Services and Distributed Programming

Distributed programming is like network programming—only the audience is different. The point of network programming is to let a human control a computer across the network. The point of distributed programming is to let computers communicate between themselves.

Humans use networking software to get data and use algorithms they don't have on their own computers. With distributed programming, automated programs can get in on this action. The programs are (one hopes) designed for the ultimate benefit of humans, but an end user doesn't see the network usage or even neccessarily know that it's happening.

The simplest and most common form of distributed programming is the web service. Web services work on top of HTTP: they generally involve sending an HTTP request to a certain URL (possibly including an XML document), and getting a response in the form of another XML document. Rather than showing this document to an end user the way a web browser would, the web service client parses the XML response document and does something with it.

We start the chapter with a number of recipes that show how to provide and use web services. We include generic recipes like Recipe 16.3, and recipes for using specific, existing web services like Recipes 16.1, 16.6, and 16.9. The specific examples are useful in their own right, but they should also help you see what kind of features you should expose in your own web services.

There are three main approaches to web services: REST-style services,[1] XML-RPC, and SOAP. You don't need any special tools to offer or use REST-style services. On the client end, you just need a scriptable web client (Recipe 14.1) and an XML parser (Recipes 11.2 and 11.3). On the server side, you just write a web application that knows how to generate XML (Recipe 11.9). We cover some REST philosophy while exploring useful services in Recipe 16.1 and Recipe 16.2.

[1] Why am I saying "REST-style" instead of REST? Because REST *is* a design philosophy, not a technology standard. REST basically says: use the technologies of the web the way they were designed to work. A lot of so-called "REST Web Services" fall short of the REST philosophy in some respect (the Amazon web service, covered in Recipe 16.1, is the most famous example). These might more accurately be called "HTTP+XML" services, or "HTTP+POX" (Plain Old XML) services. Don't get too hung up on the exact terminology.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

REST *is* HTTP; XML-RPC and SOAP are protocols that run on top of HTTP. We've devoted several recipes to Ruby's SOAP client: Recipes 16.4 and 16.7 are the main ones. Ruby's standalone SOAP server is briefly covered in Recipe 16.5. Rails provides its own SOAP server (Recipe 15.18), which incidentally also acts as an XML-RPC server.

XML-RPC isn't used much nowadays, so we've just provided a client recipe (Recipe 16.3). If you want to write a standalone XML-RPC server, check out the documentation at http://www.ntecs.de/projects/xmlrpc4r/server.html.

You can use a web service to store data on a server or change its state, but web service clients don't usually use the server to communicate with *each other*. Web services work well when there's a server with some interesting data and many clients who want it. It works less well when you want to get multiple computers to cooperate, or distribute a computation across multiple CPUs.

This is where DRb (Distributed Ruby) comes in. It's a network protocol that lets Ruby programs share objects, even when they're running on totally different computers. We cover a number of the possibilities, from simple data structure sharing (Recipe 16.10) to a networked application (Recipe 16.18) that, after the initial connection, has no visible networking code at all.

Distributed programming with DRb is a lot like multithreaded programming, except the "threads" are actually running on multiple computers. This can be great for performance. On a single CPU, multithreading makes it look like two things are happening at once, but it's just an illusion. Run two "threads" on different computers, and you can actually do twice as much work in the same time. You just need to figure out a way to split up the work and combine the results.

That's the tricky part. When you start coordinating computers through DRb, you'll run into concurrency problems and deadlock: the same problems you encounter when you share data structures between threads. You can address these problems using the same techniques that worked in Recipes 20.4 and 20.11. You'll also encounter brand new problems, like the tendency of machines to drop off the network at unfortunate times. These are more troublesome, and the solutions usually depend on the specific tasks you've assigned the machines. Recipe 16.10, the first DRb recipe, provides a brief introduction to these problems.

Recipe 16.1. Searching for Books on Amazon

Problem

You want to incorporate information about books or other cultural artifacts into your application.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

Amazon.com exposes a web service that gives you access to all kinds of information on books, music, and other media. The third-party Ruby/Amazon library provides a simple Ruby interface to the Amazon web service.

Here's a simple bit of code that searches for books with Ruby/Amazon, printing their new and used prices.

```
require 'amazon/search'

$AWS_KEY = 'Your AWS key goes here' # See below.

def price_books(keyword)
  req = Amazon::Search::Request.new($AWS_KEY)
  req.keyword_search(keyword, 'books', Amazon::Search::LIGHT) do |product|

    newp = product.our_price || 'Not available'
    usedp = product.used_price || 'not available'
    puts "#{product.product_name}: #{newp} new, #{usedp} used."
  end
end

price_books('ruby cookbook')
# Ruby Cookbook (Cookbooks (O'Reilly)): $31.49 new, not available used.
# Rails Cookbook (Cookbooks (O'Reilly)): $25.19 new, not available used.
# Ruby Ann's Down Home Trailer Park Cookbook: $10.85 new, $3.54 used.
# Ruby's Low-Fat Soul-Food Cookbook: Not available new, $12.43 used.
# …
```

To save bandwidth, this code asks Amazon for a "light" set of search results. The results won't include things like customer reviews.

Discussion

What's going on here? In one sense, it doesn't matter. Ruby/Amazon gives us a Ruby method that somehow knows about books and their Amazon prices. It's getting its information from a database somewhere, and all we need to know is how to query that database.

In another sense, it matters a lot, because this is just one example of a REST-style web service. By looking under the cover of the Amazon web services, you can see how to use other REST-style services like the ones provided by Yahoo! and Flickr.

REST-style web services operate directly on top of HTTP. Each URL in a REST system designates a resource or a set of them. When you call `keyword_search`, Ruby/ Amazon retrieves a URL that looks something like this:

```
http://xml.amazon.com/onca/xml3?KeywordSearch=ruby+cookbook&mode=books…
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

This URL designates a set of Amazon book records that match the keywords "ruby cookbook". Ruby/Amazon uses the `Net::HTTP` library to send a GET request to this URL. Amazon returns a representation of the resource, an XML document that looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<ProductInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xml.amazon.com/schemas3/dev-lite.xsd">
…
<TotalResults>11</TotalResults>
 <TotalPages>2</TotalPages>

 <Details url="http://www.amazon.com/exec/obidos/ASIN/0596523696/">
   <ProductName>Ruby Cookbook</ProductName>
   <Catalog>Book</Catalog>
   <Authors>
     <Author>Lucas Carlson</Author>
     <Author>Leonard Richardson</Author>
   </Authors>
   <ReleaseDate>September, 2006</ReleaseDate>
   <Manufacturer>O'Reilly Media</Manufacturer>
…
   </Details>
…
</ProductInfo>
```

Ruby/Amazon uses REXML to parse this XML data and turn it into `Amazon::Product` objects. An `Amazon::Product` is a lot like a Ruby `Struct`: it's got a bunch of member methods for getting information about the object (you can list these methods by calling `Product#properties`). All that information is derived from the original XML.

A REST web service works like a web site designed for a software program instead of a human. The web is good for publishing and modifying documents, so REST clients make HTTP GET requests to retrieve data, and POST requests to modify server state, just like you'd do from a web browser with an HTML form. XML is good for describing documents, so REST servers usually give out XML documents that are easy to read and parse.

How does REST relate to other kinds of web services? REST is a distinct design philosophy, but not all "REST-style" web services take it as gospel.[2] There's a sense in which "REST" is a drive for simpler web services, a reaction to the complexity of SOAP and the WS-[3] standards. There's no reason why you can't use SOAP in accordance with the REST philosophy, but in practice that never seems to happen.

[2] Amazon's web services are a case in point. They use GET requests exclusively, even when they're modifying data like the items in a shopping cart. This is very unRESTful because "put the Ruby Cookbook in my shopping cart" is a command, not an object the way a set of books is an object. To avoid the wrath of the pedant I refer to Amazon Web Services as a "REST-style" service. It would be more RESTful to define a separate resource (URL) for the shopping cart, and allow the client to POST a message to that resource saying "Hey, shopping cart, add the Ruby Cookbook to yourself."

[3] Amazon's web services are a case in point. They use GET requests exclusively, even when they're modifying data like the items in a shopping cart. This is very unRESTful because "put the Ruby Cookbook in my shopping cart" is a command, not an object the way a set of books is an object. To avoid the wrath of the pedant I refer to Amazon Web Services as a "REST-style" service. It would be more RESTful to define a separate resource (URL) for the shopping cart, and allow the client to POST a message to that resource saying "Hey, shopping cart, add the Ruby Cookbook to yourself."

Like REST, XML-RPC and SOAP web services run atop HTTP.[4] But while REST services expect clients to operate on a large URL space, XML-RPC and SOAP services are generally

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

bound to a single "server" URL. If you have a "resource" to specify, you include it in the document you send to the server. REST, XML-RPC, and SOAP all serve XML documents, but XML-RPC and SOAP serve serialized versions of data structures, and REST usually serves RDF, Atom, or Plain Old XML.

[4] SOAP services can run over other protocols, like email. But almost everyone uses HTTP. After all, they're "web services," not "Internet services."

If there were no Ruby/Amazon library, it wouldn't be hard to do the work yourself with `Net::HTTP` and REXML. It'd be more difficult to write a Ruby XML-RPC client without `xmlrpc4r`, and *much* more difficult to write a SOAP client without `SOAP::RPC::Driver`.

The downside of this flexibility is that, at least for now, every REST service is different. Everyone arranges their resources differently, and everyone's response documents need to be parsed with different code. Ruby/Amazon won't help you at all if you want to use some other REST service: you'll need to find a separate library for that service, or write your own using `Net::HTTP` and REXML.

See Also

- Like Google's web services and others, Amazon's can only be used if you sign up for an identifying key. You can sign up for an AWS key at the Amazon Web Services site (http://www.amazon.com/gp/browse.html?node=3435361)
- Get Ruby/Amazon at http://www.caliban.org/ruby/ruby-amazon.shtml: you can download it as a tarball and run `setup.rb` to install it; the same site hosts generated RDoc for the library; see especially http://www.caliban.org/ruby/ruby-amazon/classes/Amazon.html
- The Amazon Web Services documentation (http://www.amazon.com/gp/browse.html/103-8028883-0351026?node=3435361)
- Recipe 11.2, "Extracting Data from a Document's Tree Structure"
- Recipe 14.1, "Grabbing the Contents of a Web Page"
- Recipe 16.2, "Finding Photos on Flickr"
- Recipe 16.4, "Writing a SOAP Client"

Recipe 16.2. Finding Photos on Flickr

Problem

You want to use Ruby code to find freely reusable photos: perhaps to automatically illustrate a piece of text.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

The Flickr photo-sharing web site has a huge number of photos and provides web services for searching them. Many of the photos are licensed under Creative Commons licenses, which give you permission to reuse the photos under various restrictions.

There are several Ruby bindings to Flickr's various web service APIs, but its REST API is so simple that I'm just going to use it directly. Given a tag name (like "elephants"), this code will find an appropriate picture, and return the URL to a thumbnail version of the picture.

First, a bit of setup. As with Amazon and Google, to use the Flickr API at all you'll need to sign up for an API key (see below for details).

```
require 'open-uri'
require 'rexml/document'
require 'cgi'

FLICKR_API_KEY = 'Your API key here'
```

The first method, `flickr_call`, sends a generic query to Flickr's REST web service. It doesn't do anything special: it just makes an HTTP GET request and parses the XML response.[5]

[5] Some of Flickr's APIs let you do things like upload photos and add comments. You'll need to use POST requests to make these calls, since they modify the state of the site. More importantly, you'll also need to authenticate against your Flickr account.

```
def flickr_call(method_name, arg_map={}.freeze)
  args = arg_map.collect {|k,v| CGI.escape(k) + '=' + CGI.escape(v)}.join('&')
  url = "http://www.flickr.com/services/rest/?api_key=%s&method=%s&%s" %
    [FLICKR_API_KEY, method_name, args]
    doc = REXML::Document.new(open(url).read)
end
```

Now comes `pick_a_photo`, a method that uses `flickr_call` to invoke the `flickr.photos.search` web service method. That method returns a REXML `Document` object containing a `<photo>` element for each photo that matched the search criteria. I use XPath to grab the first `<photo>` element, and pass it into `small_photo_url` (defined below) to turn it into an image URL.

```
def pick_a_photo(tag)
  doc = flickr_call('flickr.photos.search', 'tags' => tag, 'license' => '4',
                    'per_page' => '1')
  photo = REXML::XPath.first(doc, '//photo')
  small_photo_url(photo) if photo
end
```

Finally, I'll define the method, `small_photo_url`. Given a `<photo>` element, it returns the URL to a smallish version of the appropriate Flickr photo.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
def small_photo_url(photo)
 server, id, secret = ['server', 'id', 'secret'].collect do |field|
   photo.attribute(field)
 end
 "http://static.flickr.com/#{server}/#{id}_#{secret}_m.jpg"
end
```

Now I can find an appropriate photo for any common word (Figure 16-1):

```
pick_a_photo('elephants')
# => http://static.flickr.com/32/102580480_506d5865d0_m.jpg

pick_a_photo('what-will-happen-tomorrow')
# => nil
```

Figure 16-1. A photo of elephants by Nick Scott-Smith

Discussion

It's nice if there's a predefined Ruby binding available for a particular REST-style web service, but it's usually pretty easy to roll your own. All you need to do is to craft an HTTP request and figure out how to process the response document. It's usually an XML document, and a well-crafted XPath statement should be enough to grab the data you want.

Note the clause `license=4` in `pick_a_photo`'s arguments to `flickr_call`. I wanted to find a picture that I could publish in this book, so I limited my search to pictures made available under a Creative Commons "Attribution" license. I can reproduce that picture of the elephants so long as I credit the person who took the photo. (Nick Scott-Smith of London. Hi, Nick!)

Flickr has a separate API call that lists the available licenses (`flickr.licenses.getInfo`), but once I looked them up and found that "Creative Commons Attribution" was number four, it was easier to hardcode the number than to look it up every time.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- The first few recipes in Chapter 11 demonstrate different ways of extracting data from XML documents; XPath (Recipe 11.4) and Rubyful Soup (Recipe 11.5) let you extract data without writing much code
- Recipe 14.1, "Grabbing the Contents of a Web Page"
- Sign up for a Flickr API key at http://www.flickr.com/services/api/key.gne
- Flickr provides REST, XML-RPC, and SOAP interfaces, and comprehensive documentation of its API (http://www.flickr.com/services/api/)
- The Flickr URL documentation shows how to turn a `<photo>` element into a URL (http://www.flickr.com/services/api/misc.urls.html)
- Flickr.rb (http://redgreenblu.com/flickr/; available as the `flickr` gem), the libyws project (http://rubyforge.org/projects/libyws; check out from CVS repository), and rflickr (http://rubyforge.org/projects/rflickr/; available as the `rflickr` gem)
- A brief explanation of the Creative Commons licences (http://creativecommons.org/about/licenses/meet-the-licenses)

Recipe 16.3. Writing an XML-RPC Client

Credit: John-Mason Shackelford

Problem

You want to call a remote method through the XML-RPC web service protocol.

Solution

Use Michael Neumann's `xmlrpc4r` library, found in Ruby's standard library.

Here's the canonical simple XML-RPC example. Given a number, it looks up the name of a U.S. state in an alphabetic list:

```
require 'xmlrpc/client'
server = XMLRPC::Client.new2('http://betty.userland.com/RPC2')
server.call('examples.getStateName', 5) # => "California"
```

Discussion

XML-RPC is a language-independent solution for distributed systems that makes a simple alternative to SOAP (in fact, XML-RPC is an ancestor of SOAP). Although it's losing ground

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

to SOAP and REST-style web services, XML-RPC is still used by many blogging engines and popular web services, due to its simplicity and relatively long history.

A XML-RPC request is sent to the server as a specially-formatted HTTP POST request, and the XML-RPC response is encoded in the HTTP response to that request. Since most firewalls allow HTTP traffic, this has the advantage (and disadvantage) that XML-RPC requests work through most firewalls. Since XML-RPC requests are POST requests, typical HTTP caching solutions (which only cache GETs) can't be used to speed up XML-RPC requests or save bandwidth.

An XML-RPC request consists of a standard set of HTTP headers, a simple XML document that encodes the name of a remote method to call, and the parameters to pass to that method. The `xmlrpc4r` library automatically converts between most XML-RPC data types and the corresponding Ruby data types, so you can treat XML-RPC calls almost like local method calls. The main exceptions are date and time objects. You can pass a Ruby `Date` or `Time` object into an XML-RPC method that expects a `dateTime.iso8601` parameter, but a method that returns a date will always be represented as an instance of `XMLRPC::DateTime`.

Table 16-1 lists the supported data types of the request parameters and the response.

Table 16-1. Supported data types

XML-RPC data type	Description	Ruby equivalent
int	Four-byte signed integer	Fixnum or Bignum
boolean	0 (false) or 1 (true)	TrueClass or FalseClass
string	Text or encoded binary data; only the characters < and & are disallowed and rendered as HTML entities	String
double	Double-precision signed floating point number	Float
dateTime.iso8601	Date/time in the format YYYYMMDDTHH:MM:SS (where T is a literal)	XMLRPC::DateTime
base64	base64-encoded binary data	String
struct	An unordered set of key value pairs where the name is always a String and the value can be any XML-RPC data type, including netsted a nested struct or array	Hash
array	A series of values that may be of any of XML-RPC data type, including a netsted struct or array; multiple data types can be used in the context of a single array	Array

Note that `nil` is not a supported XML-RPC value, although some XML-RPC implementations (including `xmlrpc4r`) follow an extension that allows it.

An XML-RPC response is another XML document, which encodes the return value of the remote method (if you're lucky) or a "fault" (if you're not). `xmlrpc4r` parses this document and transforms it into the corresponding Ruby objects.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

If the remote method returned a fault, `xmlrpc4r` raises an `XMLRPC::FaultException`. A fault contains an integer value (the fault code) and a string containing an error message. Here's an example:

```
begin
  server.call('noSuchMethod')
rescue XMLRPC::FaultException => e
  puts "Error: fault code #{e.faultCode}"
  puts e.faultString
end
# Error: fault code 7
# Can't evaluate the expression because the name "noSuchMethod" hasn't been defined.
```

Here's a more interesting XML-RPC example that searches an online UPC database:

```
def lookup_upc(upc)
  server = XMLRPC::Client.new2('http://www.upcdatabase.com/rpc')
  begin
    response = server.call('lookupUPC', upc)
    return response['found'] ? response : nil
  rescue XMLRPC::FaultException => e
    puts "Error: "
    puts e.faultCode
    puts e.faultString
  end
end

product = lookup_upc('018787765654')
product['description']                  # => "Dr Bronner's Peppermint Oil Soap"
product['size']                         # => "128 fl oz"

lookup_upc('no such UPC')               # => nil
```

See Also

- Michael Neumann's xmlrpc4r—HOWTO (http://www.ntecs.de/projects/xmlrpc4r/howto.html)
- The XML-RPC Specification (http://www.xmlrpc.com/spec)
- The extension to XML-RPC that lets it represent `nil` values (http://ontosys.com/xml-rpc/extensions.php)
- The *Ruby Developer's Guide*, published by Syngress and edited by Michael Neumann, contains over 20 pages devoted to implementing XML-RPC clients and servers with `xmlrpc4r`.
- Recipe 15.8, "Creating a Login System," shows how to serve XML-RPC requests from within a Rails application

Recipe 16.4. Writing a SOAP Client

Credit: Kevin Marshall

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You need to call a remote method through a SOAP-based web service.

Solution

Use the SOAP RPC Driver in the Ruby standard library.

This simple program prints a quote of the day. It uses the SOAP RPC Driver to connect to the SOAP web service at codingtheweb.com.

```
require 'soap/rpc/driver'
driver = SOAP::RPC::Driver.new(
            'http://webservices.codingtheweb.com/bin/qotd',
            'urn:xmethods-qotd')
```

Once the driver is set up, we define the web service method we want to call (`getQuote`). We can then call it like a normal Ruby method and display the result:

```
driver.add_method('getQuote')

puts driver.getQuote
# The holy passion of Friendship is of so sweet and steady and
# loyal and enduring a nature that it will last through a whole
# lifetime, if not asked to lend money.
# Mark Twain (1835 - 1910)
```

Discussion

SOAP is a heavyweight protocol for web services, a distant descendant of XML-RPC. As with XML-RPC, a SOAP client sends an XML representation of a method call to a server, and gets back an XML representation of a return value. The whole process is more complex than XML-RPC, but Ruby's built-in SOAP library handles the low-level details for you, leaving you free to focus on using the results in your program.

There are only a few things you need to know to build useful SOAP clients (as I run through them, I'll build another SOAP client; this one is to get stock quotes):

1. The location of the web service (known as the endpoint URL) and the namespace used by the service's documents.

```
require 'soap/rpc/driver'
driver = SOAP::RPC::Driver.new(
   'http://services.xmethods.net/soap/',        # The endpoint url
   'urn:xmethods-delayed-quotes')               # The namespace
```

2. The name of the SOAP method you want to call, and the names of its parameters.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
driver.add_method('getQuote', 'symbol')
```

Behind the scenes, that call to `add_method` actually defines a new method on the `SOAP::RPC::Driver` object. The SOAP library uses metaprogramming to create custom Ruby methods that act like SOAP methods.

3. The details about the results you expect back.

```
puts 'Stock price: %.2f' % driver.getQuote('TR')
# Stock price: 28.78
```

We expect the stock quote service in the example to return a floating-point value, which we simply display. With more complex result sets, you'll probably assign the results to a variable, which you'll treat as an array or class instance.

See Also

- Recipe 16.6, "Searching the Web with Google's SOAP Service," provides a more complex example
- Recipe 16.7, "Using a WSDL File to Make SOAP Calls Easier"

Recipe 16.5. Writing a SOAP Server

Credit: Kevin Marshall

Problem

You want to host a SOAP-based web service using a standalone server (that is, not as part of a Rails application).

Solution

Building your own SOAP server really only requires three simple steps:

1. Subclass the `SOAP::StandaloneServer` class. In the constructor, register the methods you want to expose and the arguments they should take. Here we expose a method `sayhelloto` method that expects one parameter, `username`:

```
require 'soap/rpc/standaloneServer'

class MyServer < SOAP::RPC::StandaloneServer
```

```
  def initialize(*args)
    super
    add_method(self, 'sayhelloto', 'username')
  end
end
```

2. Define the methods you exposed in step 1:

```
class MyServer
  def sayhelloto(username)
    "Hello, #{username}."
  end
end
```

3. Finally, set up and start your server. Our example server runs on port 8888 on localhost. Its name is "CoolServer" and its namespace is "urn:mySoapServer":

```
server = MyServer.new('CoolServer','urn:mySoapServer','localhost',8888)
trap('INT') { server.shutdown }
server.start
```

   We trap interrupt signals so that we can stop our server from the command line.

Discussion

We've now built a complete SOAP server. It uses the SOAP `StandaloneServer` and hosts one simple `sayhelloto` method that can be accessed at "http://localhost:8888/sayhelloto" with a namespace of "urn:mySoapServer".

To test your service, start your server in one Ruby session and then use the simple script below in another Ruby session to call the method it exposes:

```
require 'soap/rpc/driver'
driver = SOAP::RPC::Driver.new('http://localhost:8888/', 'urn:mySoapServer')
driver.add_method('sayhelloto', 'username')
driver.sayhelloto('Kevin')                              # => "Hello, Kevin."
```

See Also

- Recipe 15.18, "Exposing Web Services on Your Web Site," shows how to use the XML-RPC/SOAP server that comes with Rails
- For information on building web service *clients*, see Recipes 16.2 through 16.3, 16.4 and 16.7.
- *Ruby on Rails* by Bruce A. Tate and Curt Hibbs (O'Reilly)

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 16.6. Searching the Web with Google's SOAP Service

Problem

You want to use Google's web services to perform searches and grab their results within your Ruby application.

Solution

Google exposes a SOAP API to its search functionality, and some other miscellaneous methods like spellcheck. Call these methods with the SOAP client that comes with Ruby's standard library:

```
$KCODE = 'u' # This lets us parse UTF characters
require 'soap/wsdlDriver'

class Google
  @@key = 'JW/JqyXMzCsv7k/dxqR9E9HF+jiSgbDL'
# Get a key at http://www.google.com/apis/
  @@driver = SOAP::WSDLDriverFactory.
    new('http://api.google.com/GoogleSearch.wsdl').create_rpc_driver

  def self.search(query, options={})
     @@driver.doGoogleSearch(
       @@key,
       query,
       options[:offset] || 0,
       options[:limit] || 10,           # Note that this value cannot exceed 10
       options[:filter] || true,
       options[:restricts] || ' ',
       options[:safe_search] || false,
       options[:lr] || ' ',
       options[:ie] || ' ',
       options[:oe] || ' '
     )
   end

   def self.count(query, options={})
     search(query, options).estimatedTotalResultsCount
   end

  def self.spell(phrase)
    @@driver.doSpellingSuggestion(@@key, phrase)
  end
end
```

Here it is in action:

```
Google.count "Ruby Cookbook site:oreilly.com"
# => 368

results = Google.search "Ruby Cookbook site:oreilly.com", :limit => 7
results.resultElements.size
# => 7

results.resultElements.first["title"]
# => "oreilly.com -- Online Catalog: <b>Ruby Cookbook</b>…"

results.resultElements.first["URL"]
# => "http://www.oreilly.com/catalog/rubyckbk/"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
results.resultElements.first["snippet"]
# => "The <b>Ruby Cookbook</b> is a new addition to …"

Google.spell "tis is te centence"
# => "this is the sentence"
```

Discussion

Each of the options defined in `Google.search` corresponds to an option in the Google search API.

Table 16-2.

Name	Description
key	Unique key provided when you sign up with Google's web services.
query	The search query.
limit	How many results to grab; the maximum is 10.
offset	Which result in the list to start from.
filter	Whether or not to let Google group together similar results.
restricts	Further restrict search results to those containing this string.
safe_search	Whether or not to enable the SafeSearch filtering feature.
lr	Language restriction: lets you search for pages in specific languages.
ie	Input encoding: lets you choose the character encoding for the query.
oe	Output encoding: lets you choose the character encoding for the returned results.

See Also

- For a simpler API, see Recipe 16.7, "Using a WSDL File to Make SOAP Calls Easier"
- http://www.google.com/apis/reference.html
- http://www.google.com/help/refinesearch.html

Recipe 16.7. Using a WSDL File to Make SOAP Calls Easier

Credit: Kevin Marshall

Problem

You need to create a client for a SOAP-based web service, but you don't want to type out the definitions for all the SOAP methods you'll be calling.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

Most web services provide a WSDL file: a machine-readable description of the methods they offer. Ruby's SOAP WSDL Driver can parse a WSDL file and make the appropriate methods available automatically.

This code uses the *xmethods.com* SOAP web service to get a stock price. In Recipe 16.7, we defined the `getQuote` method manually. Here, its name and signature are loaded from a hosted WSDL file. You still have to know that the method is called `getQuote` and that it takes one string, but you don't have to write any code telling Ruby this.

```
require 'soap/wsdlDriver'
wsdl = 'http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl'
driver = SOAP::WSDLDriverFactory.new(wsdl).create_rpc_driver

puts "Stock price: %.2f" % driver.getQuote('TR')
# Stock price: 28.78
```

Discussion

According to the World Wide Web Consortium (W3), "WSDL service definitions provide documentation for distributed systems and serve as a recipe for automating the details involved in applications communication."

What this means to you is that you don't have to tell Ruby which methods a web service provides, and what arguments it expects. If you feed a WSDL file in to the Driver Factory, Ruby will give you a `Driver` object with all the methods already defined.

There are only a few things you need to know to build useful SOAP clients with a WSDL file. I'll illustrate with some code that performs a Google search and prints out the results.

1. Start with the URL to the WSDL file:

```
require 'soap/wsdlDriver'
wsdl = 'http://api.google.com/GoogleSearch.wsdl'
driver = SOAP::WSDLDriverFactory.new(wsdl).create_rpc_driver
```

2. Next you need the name of the SOAP method you want to call, and the expected types of its parameters:

```
my_google_key = 'get yours from https://www.google.com/accounts'
my_query = 'WSDL Ruby'
XSD::Charset.encoding = 'UTF8'
result = driver.doGoogleSearch(my_google_key, my_query, 0, 10, false,
                               '', false, '', '', '')
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Without WSDL, you need to tell Ruby that methods a web service exposes, and what parameters it takes. With WSDL, Ruby loads this information from the WSDL file. Of course, *you* still need to know this information so you can write the method call. In this case, you'll also need to sign up for an API key that lets you use the web service.

The Google search service returns data encoded as UTF-8, which may contain special characters that cause mapping problems to Ruby strings. That's what the call to `XSD::Charset.encoding = 'UTF8'` is for. The Soap4r and WSDL Factory libraries rely on the XSD library to handle the data type conversions from web services to native Ruby types. By explicitly telling Ruby to use UTF-8 encoding, you'll ensure that any special characters are properly escaped within your results so you can treat them as proper Ruby Strings.

```
result.class
# => SOAP::Mapping::Object

(result.methods - SOAP::Mapping::Object.instance_methods).sort
# => ["directoryCategories", "directoryCategories=", "documentFiltering",
# …
# "searchTips", "searchTips=", "startIndex", "startIndex="]
```

3. Here's how to treat the result object you get back:

```
"Query for: #{my_query}"
# => "Query for: WSDL Ruby"
"Found: #{result['estimatedTotalResultsCount']}"
# => "Found: 159000"
"Query took about %.2f seconds" % result['searchTime']
# => "Query took about 0.05 seconds"

result["resultElements"].each do |rec|
  puts "Title: #{rec["title"]}"
  puts "URL: #{rec["URL"]}"
  puts "Snippet: #{rec["snippet"]}"
  puts
end
# Title: <b>wsdl</b>: <b>Ruby</b> Standard Library Documentation
# URL: http://www.ruby-doc.org/stdlib/libdoc/wsdl/rdoc/index.html
# Snippet: #<SOAP::Mapping::Object:0xb705f560>
#
# Title: how to make SOAP4R read <b>WSDL</b> files?
# URL: http://www.ruby-talk.org/cgi-bin/scat.rb/ruby/ruby-talk/37623
# Snippet: Subject: how to make SOAP4R read <b>WSDL</b> files? <b>…</b>
# …
```

We expect the Google search service to return a complex SOAP type. The XSD library will convert it into a Ruby hash, containing some keys like `EstimatedTotalResultsCount` and `resultElements`—the latter points to an array of search results. Every search result is itself a complex type, and XSD maps it to a hash as well: a hash with keys like `snippet` and `URL`.

See Also

- Recipe 16.4, "Writing a SOAP Client," provides a more generic example of a SOAP client
- Recipe 16.6, "Searching the Web with Google's SOAP Service," shows what searching Google would be like without WSDL
- https://www.google.com/accounts to get an access key to Google Web APIs

Recipe 16.8. Charging a Credit Card

Problem

You want to charge a credit card from within your Ruby application.

Solution

To charge credit cards online, you need an account with a credit card merchant. Although there are many to choose from, Authorize.Net is one of the best and most widely used. The `payment` library encapsulates the logic of making a credit card payments with Authorize.Net, and soon it will support other gateways as well. It's available as the `payment` gem.

```
require 'rubygems'
require 'payment/authorize_net'

transaction = Payment::AuthorizeNet.new(
  :login           => 'username',
  :transaction_key => 'my_key',
  :amount          => '49.95',
  :card_number     => '4012888818888',
  :expiration      => '0310',
  :first_name      => 'John',
  :last_name       => 'Doe'
)
```

The `submit` method sends a payment request. If there's a problem with your payment (probably due to an invalid credit card), the `submit` method will raise a `Payment::PaymentError`:

```
begin
  transaction.submit
  puts "Card processed successfully: #{transaction.authorization}"
rescue Payment::PaymentError
  puts "Card was rejected: #{transaction.error_message}"
end
# Card was rejected: The merchant login ID or password is invalid
# or the account is inactive.
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

Some of the information sent during initialization of the `Payment::AuthorizeNet` class represent your account with Authorize.Net, and will never change (at least, not for the lifetime of the account). You can store this information in a YAML file called *.payment.yml* in your home directory, and have the payment library load it automatically. A *.payment.yml* file might look like this:

```
login: username
transaction_key: my_key
```

That way you don't have to hardcode `login` and `transaction_key` within your Ruby code.

If you're using the payment library from within a Rails application, you might want to put your YAML hash in the *config* directory with other configuration files, instead of in your home directory. You can override the location for the defaults file by specifying the `:prefs` key while initializing the object:

```
payment = Payment::AuthorizeNet
    .new(:prefs => "#{RAILS_ROOT}/config/payment.yml")
payment.amount = 20
payment.card_number = 'bogus'
payment.submit rescue "That didn't work"
```

Notice that after the `Payment::AuthorizeNet` object has been initialized, you can change its configuration with accessor methods.

Like most online merchants, Authorize.Net uses its own XML-formatted responses to do transactions over HTTPS. Some merchants, such as Payflow Pro, use proprietary interfaces to their backend that require a bridge with their Java or C libraries. If you're using Ruby, this approach can be cumbersome and difficult. It's worth investing some time into researching how flexible the backend is before you decide on a merchant platform for your Ruby application.

See Also

- Recipe 2.17, "Checking a Credit Card Checksum"
- The online RDoc for the `payment` library (http://payment.rubyforge.org/)
- http://authorize.net/

Recipe 16.9. Finding the Cost to Ship Packages via UPS or FedEx

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to calculate the cost to ship any item with FedEx or UPS. This is useful if you're running an online store.

Solution

FedEx and UPS provide web services that can query information on pricing as well as retrieve shipping labels. The logic for using these services has been encapsulated within the `shipping` gem:

```
require 'rubygems'
require 'shipping'

ship = Shipping::Base.new(
  :fedex_url => 'https://gatewaybeta.fedex.com/GatewayDC',
  :fedex_account => '123456789',
  :fedex_meter => '387878',

  :ups_account => '7B4F74E3075AEEFF',
  :ups_user => 'username',
  :ups_password => 'password',

  :sender_zip => 10001                  # It's shipped from Manhattan.
)

ship.weight = 2                            # It weighs two pounds.
ship.city = 'Portland'
ship.state = 'OR'
ship.zip = 97202

ship.ups.price                             # => 8.77
ship.fedex.price                           # => 5.49
ship.ups.valid_address?                    # => true
```

If you have a UPS account or a FedEx account, but not both, you can omit the account information you don't have, and instantiate a `Shipping::UPS` or a `Shipping::FedEx` object.

Discussion

You can either specify your account information during the initialization of the object (as above) or in a YAML hash. It's similar to the `payment` library described in Recipe 16.8. If you choose to use the YAML hash, you can specify the account information in a file called *.shipping.yml* within the home directory of the user running the Ruby program:

```
fedex_url: https://gatewaybeta.fedex.com/GatewayDC
fedex_account: 1234556
fedex_meter: 387878

ups_account: 7B4F74E3075AEEFF
ups_user: username
ups_password: password
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

But your directory is not a good place to keep a file being used by a Rails application. Here's how to move the *.shipping* file into a Rails application:

```
ship = Shipping::FedEx.new(:prefs => "#{RAILS_ROOT}/config/shipping.yml")
ship.sender_zip = 10001
ship.zip = 97202
ship.state = 'OR'
ship.weight = 2

ship.price > ship.discount_price                    # => true
```

Notice the use of `ship.discount_price` to find the discounted price; if you have an account with FedEx or UPS, you might be eligible for discounts.

See Also

- http://shipping.rubyforge.org/
- Recipe 16.8, "Charging a Credit Card"

Recipe 16.10. Sharing a Hash Between Any Number of Computers

Credit: James Edward Gray II

Problem

You want to easily share some application data with remote programs. Your needs are as trivial as, "What if all the computers could share this hash?"

Solution

Ruby's built-in DRb library can share Ruby objects across a network. Here's a simple hash server:

```
#!/usr/local/ruby -w
# drb_hash_server.rb
require 'drb'

# Start up DRb with a URI and a hash to share
shared_hash = {:server => 'Some data set by the server' }
DRb.start_service('druby://127.0.0.1:61676', shared_hash)
puts 'Listening for connection…'
DRb.thread.join # Wait on DRb thread to exit…
```

Run this server in one Ruby session, and then you can run a client in another:

```
require 'drb'
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Prep DRb
DRb.start_service
# Fetch the shared object
shared_data = DRbObject.new_with_uri('druby://127.0.0.1:61676')

# Add to the Hash
shared_data[:client] = 'Some data set by the client'
shared_data.each do |key, value|
  puts "#{key} => #{value}"
end
# client => Some data set by the client
# server => Some data set by the server
```

Discussion

If this looks like magic, that's the point. DRb hides the complexity of distributed programming. There are some complications (covered in later recipes), but for the most part DRb simply makes remote objects look like local objects.

The solution given above may meet your needs if you're working with a single server and client on a trusted network, but applications aren't always that simple. Issues like thread-safety and security may force you to find a more robust solution. Luckily, that doesn't require too much more work.

Let's take thread-safety first. Behind the scenes, a DRb server handles each client connection in a separate Ruby thread. Ruby's `Hash` class is not automatically thread-safe, so we need to do a little extra work before we can reliably share a hash between multiple concurrent users.

Here's a library that uses delegation to implement a thread-safe hash. A `ThreadsafeHash` object delegates all its method calls to an underlying `Hash` object, but it uses a `Mutex` to ensure that only one thread (or DRb client) can have access to the hash at a time.

```
# threadsafe_hash.rb
require 'rubygems'
require 'facet/basicobject'                # For the BasicObject class
require 'thread'                           # For the Mutex class
```

We base our thread-safe hash on the `BasicObject` class in the Facets More library (available as the `facets_more` gem). A `BasicObject` is an ordinary Ruby object, except it defines no methods at all—not even the methods of `Object`. This gives us a blank slate to work from. We can make sure that every single method of `ThreadsafeHash` gets forwarded to the underlying hash, even methods like inspect, which are defined by `Object` and which wouldn't normally trigger `method_missing`.

```
# A thread-safe Hash that delegates all its methods to a real hash.
class ThreadsafeHash < BasicObject
  def initialize(*args, &block)
    @hash = Hash.new(*args, &block)   # The shared hash
    @lock = Mutex.new                 # For thread safety
  end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  def method_missing(method, *args, &block)
    if @hash.respond_to? method  # Forward Hash method calls…
      @lock.synchronize do       # but wrap them in a thread safe lock.
        @hash.send(method, *args, &block)
      end
    else
      super
    end
  end
end
```

The next step is to build a `RemoteHash` using `BlankSlate`. The implementation is trivial. Just forward method calls onto the `Hash`, but wrap each of them in a synchronization block in order to ensure only one thread can affect the object at a time.

Now that we have a thread-safe `RemoteHash`, we can build a better server:

```
#!/usr/bin/ruby -w
# threadsafe_hash_server.rb

require 'threadsafe_hash'        # both sides of DRb connection need all classes
require 'drb'
```

We begin by pulling in our `RemoteHash` library and `DRb`:

```
$SAFE = 1 # Minimum acceptable paranoia level when sharing code!
```

The `$SAFE=1` line is *critical*! Don't put any code on a network without a minimum of `$SAFE=1`. It's just too dangerous. Malicious code, like `obj.instance_eval("`rm -rf / *`")`, must be controlled. Feel free to raise `$SAFE` even higher, in fact.

```
# Start up DRb with a URI and an object to share.
DRb.start_service('druby://127.0.0.1:61676', Threadsafe.new)
puts 'Listening for connection…'
DRb.thread.join # wait on DRb thread to exit…
```

We're now ready to start the DRb service, which we do with a URI and an object to share. If you don't want to allow external connections, you may want to replace "127.0.0.1" with "localhost" in the URI.

Since DRb runs in its own threads, the final line of the server is needed to ensure that we don't exit before those threads have done their job.

Run that code, and then you can run this client code to share a hash:

```
#!/usr/bin/ruby
# threadsafe_hash_client.rb

require 'remote_hash' # Both sides of DRb connection need all classes
require 'drb'
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Prep DRb
DRb.start_service

# Fetch the shared hash
$shared_data = DRbObject.new_with_uri('druby://127.0.0.1:61676')

puts 'Enter Ruby commands using the shared hash $shared_data…'
require 'irb'
IRB.start
```

Here again we pull in the needed libraries and point DRb at the served object. We store that object in a variable so that we can continue to access it as needed.

Then, just as an example of what can be done, we enter an IRb session, allowing you to manipulate the variable any way you like. Remember, any number of clients can connect and share this hash.

Let's illustrate some sample sessions. In the first one, we add some data to the hash:

```
$ ruby threadsafe_hash_client.rb
Enter Ruby commands using the shared hash $shared_data…
irb(main):001:0> $shared_data.keys
=> []
irb(main):002:0> $shared_data[:terminal_one] = 'Hello other terminals!'
=> "Hello other terminals!"
```

Let's attach a second client and see what the two of them find:

```
$ ruby threadsafe_hash_client.rb
Enter Ruby commands using the shared hash $shared_data…
irb(main):001:0> $shared_data.keys
=> [:terminal_one]
irb(main):002:0> $shared_data[:terminal_one]
=> "Hello other terminals!"
irb(main):003:0> $shared_data[:terminal_two] = 'Is this thing on?'
=> "Is this thing on?"
```

Going back to the first session, we can see the new data:

```
irb(main):003:0> $shared_data.each_pair do |key, value|
irb(main):004:1* puts "#{key} => #{value}"
irb(main):005:1> end
terminal_one => Hello other terminals!
terminal_two => Is this thing on?
```

Notice that, as you'd hope, the DRb magic can even cope with a method that takes a code block.

See Also

- There is a good beginning tutorial for DRb at http://www.rubygarden.org/ruby?DrbTutorial

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

- There is a helpful DRb presentation by Mark Volkmann in the "Why Ruby?" repository at http://rubyforge.org/docman/view.php/251/216/DistributedRuby.pdf
- The standard library documentation for DRb can be found at http://www.ruby-doc.org/stdlib/libdoc/drb/rdoc/index.html
- For more on the internal workings of the thread-safe hash, see Recipe 8.8, "Delegating Method Calls to Another Object," and Recipe 20.4, "Synchronizing Access to an Object"
- Recipe 20.11, "Avoiding Deadlock," for another common problem with multi-threaded programming

Recipe 16.11. Implementing a Distributed Queue

Credit: James Edward Gray II

Problem

You want to use a central server as a workhorse, queueing up requests from remote clients and handling them one at a time.

Solution

Here's a method that shares a `Queue` object with clients. Clients put job objects into the queue, and the server handles them by `yielding` them to a code block. #!/usr/bin/ruby

```
#!/usr/bin/ruby
# queue_server.rb

require 'thread'           # For Ruby's thread-safe Queue
require 'drb'

$SAFE = 1                  # Minimum acceptable paranoia level when sharing code!

def run_queue(url='druby://127.0.0.1:61676')
  queue = Queue.new        # Containing the jobs to be processed

  # Start up DRb with URI and object to share
  DRb.start_service(url, queue)
  puts 'Listening for connection…'
  while job = queue.deq
    yield job
  end
end
```

Have your server call `run_queue`, passing in a code block that handles a single job. Every time one of your clients puts a job into the server queue, the server passes the job into the code block. Here's a sample code block that can handle a fast-running job ("Report") or a slow-running job ("Process"):

```
run_queue do |job|
  case job['request']
  when 'Report'
    puts "Reporting for #{job['from']}… Done."
  when 'Process'
    puts "Processing for #{job['from']}…"
    sleep 3          # Simulate real work
    puts 'Processing complete.'
  end
end
```

If we get a couple of clients sending in requests, output might look like this:

```
$ ruby queue_server.rb
Listening for connection…

Processing for Client 1…
Processing complete.
Processing for Client 2…
Processing complete.
Reporting for Client 1… Done.
Reporting for Client 2… Done.
Processing for Client 1…
Processing complete.
Reporting for Client 2… Done.
…
```

Discussion

A client for the queue server defined in the Solution simply needs to connect to the DRB server and add a mix of "Report" and "Process" jobs to the queue. Here's a client that connects to the DRb server and adds 20 jobs to the queue at random:

```
#!/usr/bin/ruby
# queue_client.rb

require 'thread'
require 'drb'

# Get a unique name for this client
NAME = ARGV.shift or raise "Usage: #{File.basename($0)} CLIENT_NAME"

DRb.start_service
queue = DRbObject.new_with_uri("druby://127.0.0.1:61676")

20.times do
  queue.enq('request' => ['Report', 'Process'][rand(2)], 'from' => NAME)
  sleep 1 # simulating network delays
end
```

Everything from Recipe 16.10 applies here. The major difference is that Ruby ships with a thread-safe Queue. That saves us the trouble of building our own.

See Also

- Recipe 16.10

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 16.12. Creating a Shared "Whiteboard"

Credit: James Edward Gray II

Problem

You want to create the network equivalent of a whiteboard. Remote programs can place Ruby objects up on the board, examine objects on the board, or remove objects from the board.

Solution

You could just use a synchronized hash (as in Recipe 16.10), but Rinda[6] provides a data structure called a `TupleSpace` that is optimized for distributed programming. It works well when you have some clients putting data on the whiteboard, and other clients processing the data and taking it down.

[6] Rinda is a companion library to DRb. It's a Ruby port of the Linda distributed computing environment, which is based on the idea of the tuplespace. It's similar to JavaSpaces.

Let's create an application that lets clients on different parts of the network translate each others' sentences, and builds a translation dictionary as they work.

It's easier to see the architecture of the server if you see the clients first, so here's a client that adds some English sentences to a shared `TupleSpace`:

```
#!/usr/bin/ruby -w
# english_client.rb
require 'drb'
require 'rinda/tuplespace'

# Connect to the TupleSpace…
DRb.start_service
tuplespace = Rinda::TupleSpaceProxy.new(
  DRbObject.new_with_uri('druby://127.0.0.1:61676')
)
```

The English client's job is to split English sentences into words and to add each sentence to the whiteboard as a tuple: [`unique id, language, words`].

```
counter = 0
DATA.each_line do |line|
  tuplespace.write([(counter += 1), 'English', line.strip.split])
end

__END__
Ruby programmers have more fun
Ruby gurus are obsessed with ducks
Ruby programmers are happy programmers
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's a second client. It creates a loop that continually reads all the English sentences from the `TupleSpace` and puts up word-for-word translations into Pig Latin. It uses `Tuplespace#read` to read English-language tuples off the whiteboard without removing them.

```
require 'drb'
require 'rinda/tuplespace'
require 'set'

DRb.start_service
tuplespace = Rinda::TupleSpaceProxy.new(
  DRbObject.new_with_uri('druby://127.0.0.1:61676')
)

# Track of the IDs of the sentences we've translated
translated = Set.new

# Continually read English sentences off of the board.
while english = tuplespace.read([Numeric, 'English', Array])
  # Skip anything we've already translated.
  next if translated.member? english.first
  translated << english.first

  # Translate English to Pig Latin.
  pig_latin = english.last.map do |word|
   if word =~ /^[aeiou]/i
      "#{word}way"
    elsif word =~ /^([^aeiouy]+)(.+)$/i
      "#{$2}#{$1.downcase}ay"
    end
  end

  # Write the Pig Latin translation back onto the board
    tuplespace.write([english.first, 'Pig Latin', pig_latin])
end
```

Finally, here's the language server: the code that exposes a `TupleSpace` for the two clients to use. It also acts as a third client of the `TupleSpace`: it continually takes non-English sentences down off of the whiteboard (using the destructive `TupleSpace#take` method) and matches them word-for-word with the corresponding English sentences (which it also removes from the whiteboard). In this way it gradually builds an English-to-Pig Latin dictionary, which it serializes to disk with YAML:

```
#!/usr/bin/ruby -w
# dictionary_building_server.rb
require 'drb'
require 'yaml'
require 'rinda/tuplespace'

# Create a TupleSpace and serve it to the world.
tuplespace = Rinda::TupleSpace.new
DRb.start_service('druby://127.0.0.1:61676', tuplespace)

# Create a dictionary to hold the terms we have seen.
dictionary = Hash.new
# Remove non-English sentences from the board.
while translation = tuplespace.take([Numeric, /^(?!English)/, Array])
  # Match each with its English equivalent.
  english = tuplespace.take([translation.first, 'English', Array])
  # Match up the words, and save the dictionary.
  english.last.zip(translation.last) { |en, tr| dictionary[en] = tr }
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  File.open('dictionary.yaml', 'w') { |file| YAML.dump(dictionary, file) }
end
```

If you run the server and then the two clients, the server will spit out a `dictionary.yaml` file that shows how much it has already learned:

```
$ ruby dictionary_building_server.rb &
$ ruby english_client.rb
$ ruby pig_latin_client.rb &

$ cat dictionary.yaml
---
happy: appyhay
programmers: ogrammerspray
Ruby: ubyray
gurus: urusgay
ducks: ucksday
obsessed: obsessedway
have: avehay
are: areway
fun: unfay
with: ithway
more: oremay
```

Discussion

Rinda's `TupleSpace` class is pretty close to the network equivalent of a whiteboard. A "tuple" is just an ordered sequence—in this case, an array of Ruby objects. A `TupleSpace` holds these sequences and provides an interface to them.

You can add sequences of objects to the `TupleSpace` using `TupleSpace#write`. Later, the same or different code can query the object using `TupleSpace#read` or `TupleSpace#take`. The only difference is that `TupleSpace#take` is destructive; it removes the object from the `TupleSpace` as it's read.

You can select certain tuples by passing `TupleSpace#read` or `TupleSpace#take` a *template* that matches the tuples you seek. A template is just another tuple. In the example code, we used templates like `[Numeric, 'English', Array]`. Each element of a tuple is matched against the corresponding element of a template with the === operator, the same operator used in Ruby `case` statements.

That particular template will match any three-element tuple whose first element is a `Numeric` object, whose second element is the literal string `'English'`, and whose third element is an `Array` object: that is, all the English sentences currently on the whiteboard.

You can create templates containing any kind of object that will work with the === operator: for instance, a `Regexp` object in a template can match against strings in a tuple. Any `nil` slot in a template is a wildcard slot that will match anything.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- The DRb presentation by Mark Volkmann in the "Why Ruby?" repository at http://rubyforge.org/docman/view.php/251/216/DistributedRuby.pdf has some material on TupleSpaces
- Clients can also choose to be notified of `TupleSpace` events; you can see an example at http://ruby-talk.org/cgi-bin/scat.rb/ruby/ruby-talk/159065

Recipe 16.13. Securing DRb Services with Access Control Lists

Credit: James Edward Gray II

Problem

You want to keep everybody in the world (literally!) from having access to your DRb service. Instead you want to control which hosts can, and cannot, connect.

Solution

Here's the simple shared hash from Recipe 16.10, only this time it's locked down with DRb's `ACL` (access control list) class:

```
#!/usr/bin/ruby
# acl_hash_server.rb

require 'drb'
require 'drb/acl'

# Setup the security--remember to call before DRb.start_service()
DRb.install_acl(ACL.new(%w{ deny all
                            allow 192.168.1.*
                            allow 127.0.0.1 } ) )
# Start up DRb with a URI and a hash to share
shared_hash = {:server => 'Some data set by the server' }
DRb.start_service("druby://127.0.0.1:61676", shared_hash)
puts 'Listening for connection…'
DRb.thread.join # Wait on DRb thread to exit…
```

Discussion

If you bind your DRb server to `localhost`, it'll only be accessible to other Ruby processes on your computer. That's not very distributed. But if you bind your DRb server to some other hostname, anyone on your local network (if you've got a local network) or anyone on the Internet at large will be able to share your Ruby objects. You're probably not feeling that generous.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

DRb's `ACL` class provides simple white/blacklist security similar to that used by the Unix `/etc/hosts.allow` and `/etc/hosts.deny` files. The `ACL` constructor takes an array of strings. The first string of a pair is always "allow" or "deny", and it's followed by the address or addresses to allow or deny access.

String addresses can include wildcards ("**"), as shown in the solution, to allow or deny an entire range of addresses. The `ACL` class also understands the term "all," and your first address should be either "deny all" or (less likely) "allow all". Subsequent entries can relax or restrict access, as needed.

In the Solution above, the default is to deny access. Exceptions are carved out afterwards for anyone on the local IP network (192.168.1.**) and anyone on the same host as the server itself (127.0.0.1). A public DRb server might allow access by default, and deny access only to troublesome client IPs.

See Also

- Recipe 16.10, "Sharing a Hash Between Any Number of Computers"

Recipe 16.14. Automatically Discovering DRb Services with Rinda

Credit: James Edward Gray II

Problem

You want to distribute Ruby code across your local network without hardcoding the clients with the addresses of the servers.

Solution

Using Ruby's standard Rinda library, it's easy to provide zero-configuration networking for clients and services. With Rinda, machines can discover `DRb` services without providing any addresses. All you need is a running `RingServer` on the local network:

```
#!/usr/bin/ruby
# rinda_server.rb

require 'rinda/ring'       # for RingServer
require 'rinda/tuplespace' # for TupleSpace

DRb.start_service

# Create a TupleSpace to hold named services, and start running.
Rinda::RingServer.new(Rinda::TupleSpace.new)

DRb.thread.join
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

The `RingServer` provides automatic service detection for DRb servers. Any machine on your local network can find the local `RingServer` without knowing its address. Once it's found the server, a client can look up services and use them, not having to know the addresses of the `DRb` servers that host them.

To find the Rinda server, a client broadcasts a UDP packet asking for the location of a `RingServer`. All computers on the local network will get this packet, and if a computer is running a `RingServer`, it will respond with its address. A server can use the `RingServer` to register services; a client can use the `RingServer` to look up services.

A `RingServer` object keeps a service listing in a shared `TupleSpace` (see Recipe 16.12). Each service has a corresponding tuple with four members:

- The literal symbol `:name`, which indicates that the tuple is an entry in the `RingServer` namespace.
- The symbol of a Ruby class, indicating the type of the service.
- The `DRbObject` shared by the service.
- A string description of the service.

By retrieving this `TupleSpace` remotely, you can look up services as tuples and advertise your own services. Let's advertise an object (a simple `TupleSpace`) through the `RingServer` under the name `:TupleSpace`:

```
#!/usr/bin/ruby
# share_a_tuplespace.rb

require 'rinda/ring'        # for RingFinger and SimpleRenewer
require 'rinda/tuplespace'  # for TupleSpace

DRb.start_service
ring_server = Rinda::RingFinger.primary

# Register our TupleSpace service with the RingServer
ring_server.write( [:name, :TupleSpace, Rinda::TupleSpace.new, 'Tuple Space'],
                   Rinda::SimpleRenewer.new )

DRb.thread.join
```

The `SimpleRenewer` sent in with the namespace listing lets the `RingServer` periodically check whether the service has expired.

Now we can write clients that find this service by querying the `RingServer`, without having to know which machine it lives on. All we need to know is the name of the service:

```
#!/usr/bin/ruby
# use_a_tuplespace.rb
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'rinda/ring' # for RingFinger
require 'rinda/tuplespace' # for TupleSpaceProxy

DRb.start_service
ring_server = Rinda::RingFinger.primary

# Ask the RingServer for the advertised TupleSpace.
ts_service = ring_server.read([:name, :TupleSpace, nil, nil])[2]
tuplespace = Rinda::TupleSpaceProxy.new(ts_service)

# Now we can use the object normally:
tuplespace.write([:data, rand(100)])
puts "Data is #{tuplespace.read([:data, nil]).last}."
# Data is 91.
```

These two programs locate each other without needing hardcoded IP addresses. Addresses are still being used under the covers, but the address to the Rinda server is discovered automatically through UDP, and all the other addresses are kept in the Rinda server.

`Rinda::RingFinger.primary` stores the first `RingServer` to respond to your Ruby process's UDP packet. If your local network is running more than one `RingServer`, the first one to respond might not be the one with the service you want, so you should probably only run one `RingServer` on your network. If you do have more than one `RingServer`, you can iterate over them with `Rinda::RingFinger#each`.

See Also

- Recipe 16.12, "Creating a Shared "Whiteboard"
- Recipe 16.18, "A Remote-Controlled Jukebox"
- Eric Hodel has a Rinda::RingServer tutorial at http://segment7.net/projects/ruby/drb/rinda/ringserver.html

Recipe 16.15. Proxying Objects That Can't Be Distributed

Credit: James Edward Gray II

Problem

You want to allow classes to connect to your `DRb` server, without giving the server access to the class definition. Perhaps you've given clients an API to implement, and you don't want to make everyone send you the source to their implementations just so they can connect to the server.

...OR...

You have some code that is tied to local resources: database connections, log files, or even just the closure aspect of Ruby's blocks. You want this code to interact with a DRb server, but it must be run locally.

...OR...

You want to send an object to a DRb server, perhaps as a parameter to a method; but you want the server to notice changes to that object as your local code modifies it.

Solution

Rather than sending an object to the server, you can ask DRb to send a proxy instead. When the server acts on the proxy, a description of the act will be sent across the network. The client end will actually perform the action. In effect, you've partially switched the roles of the client and the server.

You can set up a proxy in two simple steps. First, make sure your client code includes the following line before it interacts with any server objects:

```
DRb.start_service # The client needs to be a DRb service too.
```

That's generally just a good habit to get into with DRb client code, because it allows DRb to magically support some constructs (like Ruby's blocks) by sending a proxy object when necessary. If you're intentionally trying to send a proxy, it becomes essential.

As long as your client is a DRb service of its own, you can proxy all objects made from a specific class or individual objects by including the DRbUndumped module:

```
class MyLocalClass
  include DRbUndumped # The magic line. All objects of this type are proxied.
  # …
end

# … OR …

my_local_object.extend DRbUndumped # Proxy just this object.
```

Discussion

Under normal circumstances, DRb is very simple. A method call is packaged up (using Marshal) as a target object, method name, and some arguments. The resulting object is sent over the wire to the server, where it's executed. The important thing to notice is that the server receives copies of the original arguments.

The server unmarshals the data, invokes the method, packages the result, and sends it back. Again, the result objects are copied to the client.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

But that process doesn't always work. Perhaps the server needs to pass a code block into a method call. Ruby's blocks cannot be serialized. `DRb` notices this special case and sends a proxy object instead. As the server interacts with the proxy, the calls are bundled up and sent back to you, just as described above, so everything just works.

But `DRb` can't magically notice all cases where copying is harmful. That's why you need `DRbUndumped`. By extending an object with `DRbUndumped`, you can force `DRb` to send a proxy object instead of the real object, and ensure that your code stays local.

If all this sounds confusing, a simple example will probably clear it right up. Let's code up a trivial hello server:

```
#!/usr/bin/ruby
# hello_server.rb
require 'drb'

# a simple greeter class
class HelloService
  def hello(in_stream, out_stream)
    out_stream.puts 'What is your name?'
    name = in_stream.gets.strip
    out_stream.puts "Hello #{name}."
  end
end

# start up DRb with URI and object to share
DRb.start_service('druby://localhost:61676', HelloService.new)
DRb.thread.join # wait on DRb thread to exit…
```

Now we try connecting with a simple client:

```
#!/usr/bin/ruby
# hello_client.rb
require 'drb'

# fetch service object and ask it to greet us…
hello_service = DRbObject.new_with_uri('druby://localhost:61676')
hello_service.hello($stdin, $stdout)
```

Unfortunately, that yields an error message. Obviously, `$stdin` and `$stdout` are local resources that won't be available from the remote service. We need to pass them by proxy to get this working:

```
#!/usr/bin/ruby
# hello_client2.rb
require 'drb'

DRb.start_service # make sure client can serve proxy objects…
# and request that the streams be proxied
$stdin.extend DRbUndumped
$stdout.extend DRbUndumped

# fetch service object and ask it to greet us…
hello_service = DRbObject.new_with_uri('druby://localhost:61676')
hello_service.hello($stdin, $stdout)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

With that client, DRb has remote access to the streams (through the proxy objects) and can read and write them as needed.

See Also

- Recipe 16.10, "Sharing a Hash Between Any Number of Computers"
- Eric Hodel's "Introduction to DRb" covers DRbUndumped (http://segment7.net/projects/ruby/drb/introduction.html)
- The DRb presentation by Mark Volkmann in the "Why Ruby?" repository at http://rubyforge.org/docman/view.php/251/216/DistributedRuby.pdf has some material on DRbUndumped

Recipe 16.16. Storing Data on Distributed RAM with MemCached

Credit: Ben Bleything with Michael Granger

Problem

You need a lightweight, persistent storage space, and you have systems on your network that have unused RAM.

Solution

memcached provides a distributed in-memory cache. When used with a Ruby client library, it can be used to store almost any Ruby object. See the Discussion section below for more information, and details of where to get memcached.

In this example, we'll use Michael Granger's Ruby-MemCache library, available as the Ruby-MemCache gem.

Assume you have a memcached server running on the machine at IP address 10.0.1.201. You can use the memcache gem to access the cache as though it were a local hash. This Ruby code will store a string in the remote cache:

```
require 'rubygems'
require 'memcache'

MC = MemCache.new '10.0.1.201'

MC[:test] = 'This string lives in memcached!'
```

The string has been placed in your memcached with the key :test. You can fetch it from a different Ruby session:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'rubygems'
require 'memcache'

MC = MemCache.new '10.0.1.201'

MC[:test]                          # => "This string lives in memcached!"
```

You can also place more complex objects in `memcached`. In fact, any object that can be serialized with `Marshal.dump` can be placed in `memcached`. Here we store and retrieve a hash:

```
hash = {
  :roses => 'are red',
  :violets => 'are blue'
}

MC[:my_hash] = hash
MC[:my_hash][:roses]               # => "are red"
```

Discussion

`memcached` was originally designed to alleviate pressure on the database servers for LiveJournal.com. For more information about how `memcached` can be used for this kind of purpose, see Recipe 16.17.

`memcached` provides a lightweight, distributed cache space where the cache is held in RAM. This makes the cache extremely fast, and it never blocks on disk I/O. When effectively deployed, `memcached` can significantly reduce the load on your database servers by farming out storage to unused RAM on other machines.

To start using `memcached`, you'll need to download the server (see below). You can install it from source, or get it via most *nix packaging systems.

Next, find some machines on your network that have extra RAM. Install `memcached` on them, then start the daemon with this command:

```
$ memcached -d -m 1024
```

This starts up a `memcached` instance with a 1024-megabyte memory cache (you can, of course, vary the cache size as appropriate for your hardware). If you run this command on the machine with IP address 10.0.1.201, you can then access it from other machines on your local network, as in the examples above.

`memcached` also supports more advanced functions, such as conditional sets and expiration times. You can also combine multiple machines into a single virtual cache. For

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

more information about these possibilities, refer to the `memcached` documentation and to the documentation for the Ruby library that you're using.

See Also

- Recipe 13.2, "Serializing Data with Marshal"
- Recipe 16.7, "Using a WSDL File to Make SOAP Calls Easier"
- The `memcached` homepage, located at http://danga.com/memcached/, contains further information about memcached, documentation, and links to client libraries for other languages; there is also a mailing list at http://lists.danga.com/mailman/listinfo/memcached
- The Ruby-MemCache homepage is at http://deveiate.org/projects/RMemCache; if you install Ruby-MemCache from source, you'll also need to install `IO::Reactor` (http://deveiate.org/projects/IO-Reactor)
- The Robot Co-op has released their own `memcached` library, memcache-client, available at http://dev.robotcoop.com/Libraries/ or via the `memcache-client` gem; it is reported to be API-compatible with Ruby-MemCache

Recipe 16.17. Caching Expensive Results with MemCached

Credit: Michael Granger with Ben Bleything

Problem

You want to transparently cache the results of expensive operations, so that code that triggers the operations doesn't need to know how to use the cache. The `memcached` program, described in Recipe 16.16, lets you use other machines' RAM to store key-value pairs. The question is how to hide the use of this cache from the rest of your code.

Solution

If you have the luxury of designing your own implementation of the expensive operation, you can design in transparent caching from the beginning. The following code defines a `get` method that delegates to `expensive_get` if it can't find an appropriate value in the cache. In this case, the expensive operation that gets cached is the (relatively inexpensive, actually) string reversal operation:

```
require 'rubygems'
require 'memcache'

class DataLayer

  def initialize(*cache_servers)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    @cache = MemCache.new(*cache_servers)
  end

  def get(key)
    @cache[key] ||= expensive_get(key)
  end
 alias_method :[], :get

  protected
  def expensive_get(key)
    # …do expensive fetch of data for 'key'
    puts "Fetching expensive value for #{key}"
    key.to_s.reverse
  end
end
```

Assuming you've got a `memcached` server running on your local machine, you can use this `DataLayer` as a way to cache the reversed versions of strings:

```
layer = DataLayer.new( 'localhost:11211' )

3.times do
  puts "Data for 'foo': #{layer['foo']}"
end
# Fetching expensive value for foo
# Data for 'foo': oof
# Data for 'foo': oof
```

Discussion

That's the easy case. But you don't always get the opportunity to define a data layer from scratch. If you want to add memcaching to an existing data layer, you can create a caching strategy and add it to your existing classes as a mixin.

Here's a data layer, already written, that has no caching:

```
class MyDataLayer
  def get(key)
    puts "Getting value for #{key} from data layer"
    return key.to_s.reverse
  end
end
```

The data layer doesn't know about the cache, so all of its operations are expensive. In this instance, it's reversing a string every time you ask for it:

```
layer = MyDataLayer.new

"Value for 'foo': #{layer.get('foo')}"
# Getting value for foo from data layer
# => "Value for 'foo': oof"

"Value for 'foo': #{layer.get('foo')}"
# Getting value for foo from data layer
# => "Value for 'foo': oof"

"Value for 'foo': #{layer.get('foo')}"
# Getting value for foo from data layer
# => "Value for 'foo': oof"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Let's improve performance a little by defining a caching mixin. It'll wrap the `get` method so that it only runs the expensive code (the string reversal) if the answer isn't already in the cache:

```
require 'memcache'

module GetSetMemcaching
  SERVER = 'localhost:11211'

  def self::extended(mod)
    mod.module_eval do
      alias_method :__uncached_get, :get
      remove_method :get

      def get(key)
        puts "Cached get of #{key.inspect}"
        get_cache()[key] ||= __uncached_get(key)
      end
      def get_cache
        puts "Fetching cache object for #{SERVER}"
        @cache ||= MemCache.new(SERVER)
      end
    end
    super
  end

  def self::included(mod)
    mod.extend(self)
    super
  end
end
```

Once we mix `GetSetMemcaching` into our data layer, the same code we ran before will magically start to use use the cache:

```
# Mix in caching to the pre-existing class
MyDataLayer.extend(GetSetMemcaching)

"Value for 'foo': #{layer.get('foo')}"
# Cached get of "foo"
# Fetching cache object for localhost:11211
# Getting value for foo from data layer
# => "Value for 'foo': oof"

"Value for 'foo': #{layer.get('foo')}"
# Cached get of "foo"
# Fetching cache object for localhost:11211
# => "Value for 'foo': oof"

"Value for 'foo': #{layer.get('foo')}"
# Cached get of "foo"
# Fetching cache object for localhost:11211
# => "Value for 'foo': oof"
```

The examples above are missing a couple features you'd see in real life. Their API is very simple (just get methods), and they have no cache invalidation—items will stay in the cache forever, even if the underlying data changes.

The same basic principles apply to more complex caches, though. When you need a value that's expensive to find or calculate, you first ask the cache for the value, keyed by its identifying feature. The cache might map a SQL query to its result set, a primary key to the corresponding database object, an array of compound keys to the corresponding database object, and so on. If the object is missing from the cache, you fetch it the expensive way, and put it in the cache.

See Also

- The Ruby on Rails wiki has a page full of `memcached` examples at http://wiki.rubyonrails.com/rails/pages/MemCached; this should give you more ideas on how to use `memcached` to speed up your application

Recipe 16.18. A Remote-Controlled Jukebox

What if you had a jukebox on your main computer that played random or selected items from your music collection? What if you could search your music collection and add items to the jukebox queue from a laptop in another room of the house?

Ruby can help you realize this super-geek dream—the software part, anyway. In this recipe, I'll show you how to write a jukebox server that can be programmed from any computer on the local network.

The jukebox will consist of a client and a server. The server broadcasts its location to a nearby Rinda server so clients on the local network can find it without knowing the address. The client will look up the server with Rinda and then communicate with it via DRb.

What features should the jukebox have? When there are no clients interfering with its business, the server will pick random songs from a predefined playlist and play them. It will call out to external Unix programs to play songs on the local computer's audio system (if you have a way of broadcasting songs through streaming audio, say, an IceCast server, it could use that instead).

A client can query the jukebox, stop or restart it, or request that a particular song be played. The jukebox will keep requests in a queue. Once it plays all the requests, it will resume playing songs at random.

Since we'll be running subprocesses to access the sound card on the computer that runs the jukebox, the `Jukebox` object can't be distributed to another machine. Instead, we need to proxy it with `DRbUndumped`.

The first thing we need to do is start a `RingServer` somewhere on our local network. Here's a reprint of the `RingServer` program from Recipe 16.14:

```
#!/usr/bin/ruby
# rinda_server.rb

require 'rinda/ring'          # for RingServer
require 'rinda/tuplespace'    # for TupleSpace

DRb.start_service

# Create a TupleSpace to hold named services, and start running.
Rinda::RingServer.new(Rinda::TupleSpace.new)

DRb.thread.join
```

Here's the jukebox server file. First, we'll define the `Jukebox` server class, and set up its basic behavior: to play its queue and pick randomly when the queue is empty.

```
#!/usr/bin/ruby -w
# jukebox_server.rb
require 'drb'
require 'rinda/ring'
require 'rinda/tuplespace'
require 'thread'
require 'find'

DRb.start_service

class Jukebox
  include DRbUndumped
  attr_reader :now_playing, :running

  def initialize(files)
    @files = files
    @songs = @files.keys
    @now_playing = nil
    @queue = []
  end

  def play_queue
    Thread.new(self) do
      @running = true
      while @running
        if @queue.empty?
          play songs[rand(songs.size)]
        else
          play @queue.shift
        end
      end
    end
  end
```

Next, we'll write the methods that a client can use:

```
# Adds a song to the queue. Returns the new size of the queue.
def <<(song)
  raise ArgumentError, 'No such song' unless @files[song]
  @queue.push song
  return @queue.size
end

# Returns the current queue of songs.
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
def queue
  return @queue.clone.freeze
end

# Returns the titles of songs that match the given regexp.
def songs(regexp=/.*/)
  return @songs.grep(regexp).sort
end

# Turns the jukebox on or off.
def running=(value)
  @running = value
  play_queue if @running
end
```

Finally, here's the code that actually plays a song, by calling out to a preinstalled program —either `mpg123` or `ogg123`, depending on the extension of the song file:

```
  private

  # Play the given through this computer's sound system, using a
  # previously installed music player.
  def play(song)
    @now_playing = song

    path = @files[song]
    player = path[-4..path.size] == '.mp3' ? 'mpg123' : 'ogg123'
    command = %{#{player} "#{path}"}
    # The player and path both come from local data, so it's safe to
    # untaint them.
    command.untaint
    system(command)
  end
end
```

Now we can use the `Jukebox` class in a script. This one treats `ARGV` as a list of directories. We descend each one looking for music files, and feed the results into a `Jukebox`:

```
if ARGV.empty?
  puts "Usage: #{__FILE__} [directory full of MP3s and/or OGGs] …"
  exit
else
  songs = {}
  Find.find(*ARGV) do |path|
    if path =~ /\.(mp3|ogg)$/
      name = File.split(path)[1][0..-5]
      songs[name] = path
    end
  end
end

jukebox = Jukebox.new(songs)
```

So far there hasn't been much distributed code, and there won't be much total. But we do need to register the `Jukebox` object with Rinda so that clients can find it:

```
# Set safe before we start accepting connections from outside.
$SAFE = 1
puts "Registering…"
# Register the Jukebox with the local RingServer, under its class name.
ring_server = Rinda::RingFinger.primary
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
ring_server.write([:name, :Jukebox, jukebox, "Remote-controlled jukebox"],
                  Rinda::SimpleRenewer.new)
```

Start the jukebox running, and we're in business:

```
jukebox.play_queue
DRb.thread.join
```

Now we can query and manipulate the jukebox from an `irb` session on another computer:

```
require 'rinda/ring'
require 'rinda/tuplespace'

DRb.start_service
ring_server = Rinda::RingFinger.primary
jukebox = ring_server.read([:name, :Jukebox, nil, nil])[2]

jukebox.now_playing                          # => "Chickadee"
jukebox.songs(/D/)
# => ["ID 3", "Don't Leave Me Here (Over There Would Be Fine)"]

jukebox << 'ID 3'                       # => 1
jukebox << "Attack of the Good Ol' Boys from Planet Honky-Tonk"
# ArgumentError: No such song
jukebox.queue                                # => ["ID 3"]
```

But it'll be easier to use if we write a real client program. Again, there's almost no `DRb` programming in the client, which is as it should be. Once we have the remote `Jukebox` object, we can use it just like we would a local object.

First, we have some preliminary argument checking:

```
#!/usr/bin/ruby -w
# jukebox_client.rb

require 'rinda/ring'

NO_ARG_COMMANDS = %w{start stop now-playing queue}
ARG_COMMANDS = %w{grep append grep-and-append}
COMMANDS = NO_ARG_COMMANDS + ARG_COMMANDS

def usage
  puts "Usage: #{__FILE__} [#{COMMANDS.join('|')}] [ARG]"
  exit
end

usage if ARGV.size < 1 or ARGV.size > 2

command = ARGV[0]
argument = nil
usage unless COMMANDS.index(command)

if ARG_COMMANDS.index(command)
  if ARGV.size == 1
    puts "Command #{command} takes an argument."
    exit
  else
    argument = ARGV[1]
  end
  elsif ARGV.size == 2
    puts "Command #{command} takes no argument."
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    exit
  end
```

Next, the only distributed code in the client: the fetch of the `Jukebox` object from the Rinda server.

```
DRb.start_service
ring_server = Rinda::RingFinger.primary

jukebox = ring_server.read([:name, :Jukebox, nil, nil])[2]
```

Now that we have the `Jukebox` object (rather, a proxy to the real `Jukebox` object on the other computer), we can apply the user's desired command to it:

```
case command
when 'start' then
  if jukebox.running
    puts 'Already running.'
  else
    jukebox.running = true
    puts 'Started.'
  end
when 'stop' then
  if jukebox.running
    jukebox.running = false
    puts 'Jukebox will stop after current song.'
  else
    puts 'Already stopped.'
  end
when 'now-playing' then
  puts "Currently playing: #{jukebox.now_playing}"
when 'queue' then
  jukebox.queue.each { |song| puts song }
when 'grep'
  jukebox.songs(Regexp.compile(argument)).each { |song| puts song }
when 'append' then
  jukebox << argument
  jukebox.queue.each { |song| puts song }
when 'grep-and-append' then
  jukebox.songs(Regexp.compile(argument)).each { |song| jukebox << song }
  jukebox.queue.each { |song| puts song }
end
```

Some obvious enhancements to this program:

- Combine the server with the ID3 parser from Recipe 6.17 to provide more reliable title information, as well as artist and other metadata.
- Make the ID3 metadata searchable, so that you can search for songs by a particular band.
- Make the `@songs` data structure capable of handling multiple distinct songs with the same name.
- Make the selection keep track of song history, so that it doesn't choose to play the same song twice in the row.
- Have the jukebox send its selections to a program that streams audio over the network, rather than to programs that play the music locally. This way you can listen to the

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

jukebox from any computer in your house. Without this step, you need to wire your whole house for sound, or have really loud speakers, or a really small house (like mine).

See Also

- Recipe 6.17, "Processing a Binary File"
- Recipe 16.14, "Automatically Discovering DRb Services with Rinda"
- Recipe 16.15, "Proxying Objects That Can't Be Distributed"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01