j-INT-J-PARALLEL-PROG manuscript No.
(will be inserted by the editor)

dRuby and Rinda

Implementation and application of distributed Ruby and its parallel coordination mechanism

Masatoshi SEKI[1]

www.druby.org

Received: date / Revised version: date

Abstract The object-oriented scripting language Ruby is admired by many programmers for being easy to write in, and for its flexible, dynamic nature. In the last few years, the Ruby on Rails web application framework, popular for its productivity benefits, has brought about a renewed attention to Ruby for enterprise use. As the focus of Ruby has broadened from small tools and scripts, to large applications, the demands on Ruby's distributed object environment have also increased, as has the need for information about its usage, performance and examples of common practices.

dRuby and Rinda were developed by the author as the distributed object environment and shared tuplespace implementation for the Ruby language, and are included as part of Ruby's standard library. dRuby extends method calls across the network while retaining the benefits of Ruby. Rinda builds on dRuby to bring the functionality of Linda, the glue language for distributed co-ordination systems, to Ruby. This article discusses the design policy and implementation points of these two systems, and demonstrates their simplicity with sample code and examples of their usage in actual applications. In addition to dRuby and Rinda's appropriateness for prototyping distributed systems, this article will also demonstrate that dRuby and Rinda are building a reputation for being suitable infrastructure components for real-world applications.

1 Introduction

This article aims at giving you details on dRuby and Rinda. dRuby provides distributed object environment for Ruby. Rinda is a coordination mechanism running on dRuby. First, I would like to introduce Ruby. Ruby is an object-oriented scripting language created by Yukihiro Matsumoto. Until recently, Ruby's popularity was limited to a small community of early-adopter programmers. Ruby is now rapidly gathering attention amongst business users for its potential productivity gains. An English translation of the author's Japanese dRuby book is currently being planned by the Pragmatic Bookshelf.

Ruby has the following characteristics:

- Standard object-oriented features, such as classes and methods
- Everything is an object
- Untyped variables
- Easy-to-use libraries
- Simple, easy-to-learn syntax
- Garbage collector
- Rich reflection functionality
- User level threads

Ruby is categorized as a dynamic object-oriented language. Everything is composed of objects, and there is no explicit typing of variables. Unification of methods is done at execution time. Furthermore, Ruby's rich reflection functionality supports meta programming. Ruby is elaborately designed to set subtle constraints (in addition to its outstanding flexibility) that lead programmers to naturally express what they have in mind, and help them unconsciously realize the essence of object-oriented programming.

dRuby extends Ruby's method calls across processes and across the network. Rinda is an implementation of the Linda coordination model[3]. Rinda is based on dRuby and provides Ruby with tuple spaces. This article introduces not only dRuby's concept and its design policy but also its implementation and practical usage. This article discusses:

- The way of dRuby - dRuby's overview and its design policy
- dRuby's implementations
- Performance - Overhead in using dRuby
- Application - A real system executing dRuby, Rinda's overview and a real system executing Rinda

2 The way of dRuby

In this section, I describe the design policy and characteristics of dRuby.

Blaine Cook, lead developer of Twitter (a micro-blogging service), described dRuby in his presentation, "Scaling Twitter". [1]

- Stupid Easy, Reasonably Fast
- Kinda Flaky, Zero Redundancy, Tightly Coupled.

Cook's remarks are a good starting point for this discussion.

[1] http://www.slideshare.net/Blaine/scaling-twitter/

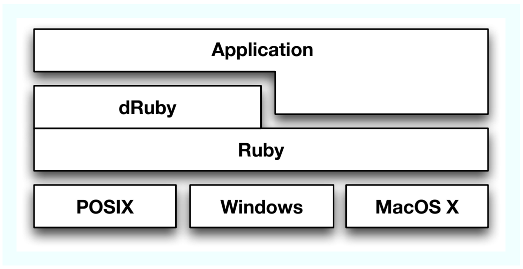

Fig. 1 dRuby application software structure

2.1 Characteristics of dRuby.

dRuby is one of several RMI libraries for Ruby. I did not aim for dRuby to be just another conventional distributed object system for Ruby. Rather, I intended to extend Ruby method calls to other processes and other machines. I think compatibility with Ruby is more important than interoperability with other distributed systems.

As a result, dRuby extends the Ruby interpreter across other processes and other machines, both in a physical sense and a temporal sense. Therefore, it is easy to "distributize" an existing Ruby script.

dRuby has the following characteristics:

– Exclusively for Ruby
– Written purely in Ruby
– No "service" specification or associated interface definition language (IDL) required

From the perspective of a conventional distributed object system, dRuby also has the following characteristics:

– Easy to set-up
– Easy to learn
– Automatic selection of object transmission strategy (pass by value or pass by reference)
– Reasonably fast
– Ruby processes act as both "clients" and "servers"—so there is no need for separate server processes.

dRuby is a distributed object system exclusively for use with Ruby. dRuby can run on any machine that runs Ruby, irrespective of operating system. dRuby can be used to exchange objects and call methods across platforms.

dRuby is written completely in Ruby, not a special extension written in C. Thanks to Ruby's excellent thread, socket and marshaling class libraries, the initial version of dRuby was implemented in just 200 lines of code. I believe that this demonstrates the power and elegance of Ruby's class libraries. dRuby is currently included as part of the standard distribution of Ruby as one of its standard libraries, so dRuby is available wherever Ruby is installed.

2.2 Compatibility with Ruby

dRuby adds a distributed object system to Ruby while preserving as much of the "feel" of Ruby as possible. Ruby programmers should find dRuby to be a comfortable, seamless extension of Ruby. Programmers accustomed to other conventional distributed object systems, however, may find dRuby to be a little strange.

Variables in Ruby are not typed, and assignment is not restricted by inheritance hierarchies. Unlike languages with statically checked variables, such as Java, objects are not checked for correctness before execution, and method look-up is only conducted at execution time (when methods are called). This is an important characteristic of the Ruby language.

dRuby operates in the same fashion. In dRuby, client stubs (the DRbObject, also called the "reference" in dRuby) are similarly not typed, and method look-up is only conducted at execution time. There is no need for a listing of exposed methods or inheritance information to be known in advance. Thus, there is no need to define an interface (e.g. by IDL).

Aside from allowing method calls across a network, dRuby has been carefully developed to adhere as closely to regular Ruby behaviour as possible. Consequently, much of unique benefits of Ruby are available for the programmer to enjoy.

For example, methods called with blocks (originally called iterators) and exceptions can be handled as if they were local. Mutex, Queue and other thread synchronization mechanisms can also be used for inter-process synchronization without any special consideration. I give an example of Queue usage in 3.1.1.

2.3 Passing Objects

.

Concepts that didn't originally exist in Ruby were introduced in dRuby as naturally as possible. Object transmission is a good example. When remote methods are called, objects such as the method arguments, return values and exceptions are transmitted. Method arguments are transmitted from client to server, while exceptions and return values are transmitted from server to client. In this article, I will refer to both of these types of object transmission as object exchange.

Assignment (or binding) to variables in Ruby is always by reference. Clones of objects are never assigned. It is, however, different in dRuby. In the world of distributed objects, distinguishing between "pass by value" and "pass by reference" is an unavoidable fact of life. This is true also of dRuby.

While a computing model where references are continually exchanged forever (or until they become nil) is conceivable, in reality applications will, at some point, need "values". The mechanism provided by dRuby minimizes the need for programmers to care about the difference between by-value or by-reference object exchange, while also striving to be reasonably efficient. That is, programmers do not need to explicitly specify whether to use pass-by-value or pass-by-reference. Instead, the system automatically decides which to use. This decision is made using a simple rule – serializable objects are passed by value, while unserializable objects are passed by reference. Methods of objects exported by pass-by-reference (and of objects explicitly exported via DRb.start_service) are invoked from a remote process via the dRuby server. If this service has not been started, an exception is raised. So, I recommend to always use DRb.start service even if no object is being explicitly exported, at least in the development phase.

Although this rule may not always be correct, in most situations it will work. Firstly, note that it is impossible for objects that cannot be serialized to be passed by value. The problematic case is where a serializable object that is more appropriately passed by reference is instead passed by value. To handle this case, dRuby provides a mechanism whereby serializable objects can be explicitly marked to be passed by reference. An example will be discussed later in this article.

By automatically choosing the means of object transmission, dRuby minimizes the amount of code that needs to be written to handle object transmission.

dRuby's lack of a need for interface definition (e.g. IDL) and declaration of object transmission style, are not the only ways that dRuby differs from other distributed object systems. This is because dRuby aims to be a "Ruby-like distributed object system", and perhaps also why dRuby may be perceived as being "kinda flaky".

2.4 Unsupported things

Finally, I shall introduce some of the features that dRuby does not support, namely garbage collection and security.

dRuby does not implement distributed garbage collection because I have not found a solution that is both cheap and realistic. Currently, it is the responsibility of the application to prevent exported objects from being garbage collected. The option to protect objects from garbage collection using a ping mechanism[2] has been provided, however, there is a risk that circular references will give rise to objects that never get garbage collected.

[2] The ping mechanism is composed of DRb::TimerIdConv and periodical access. DRb::TimerIdConv keeps objects alive for a certain amount of time after their last access. http://d.hatena.ne.jp/m_seki/20061101, http://segment7.net/projects/ruby/drb/idconv.html

Possible solutions to this problem, including the modification of the Ruby interpreter, are currently being explored.

dRuby currently does not provide any mechanisms for security. At most, dRuby imposes the same restrictions on method visibility as Ruby does, but is helpless against malicious attacks. It is, however, possible to use SSL to secure network communications.

In this chapter, I described dRuby design policy. To summarize, dRuby does extend Ruby's method calls nearly seamlessly, so dRuby is not just a standard Ruby-like interface of RMI. Therefor, it is easy to "distributize" an existing Ruby script. dRuby co-exists with traditional RMI approaches like XML-RPC, SOAP, CORBA. For example, some developers use one of these via HTTP as an interface for external network access and dRuby in their internal backend systems.

3 dRuby's Implementation

In this chapter, I discuss some interesting features of dRuby and its implementation. Using code from the initial version of dRuby and other sample code, I will describe in detail the basics of RMI usage in dRuby and the mechanism of object transmission.

3.1 Basic RMI

First, I shall explain the implementation of basic method calling using actual code.

3.1.1 An Example: The producer-consumer problem The following code is a typical implementation of the producer-consumer problem using a shared queue.

```
# shared queue server
require 'drb/drb'                                  # (1)
require 'thread'                                   # (2)

queue = SizedQueue.new(10)                         # (3)
DRb.start_service('druby://localhost:9999', queue)  # (4)
sleep                                              # (5)
```

Applications using dRuby must start by loading 'drb/drb' (1). Next, a SizedQueue object (3) with a limited number of buffer elements is instantiated. 'thread'(2) is needed for using SizedQueue class. Then the dRuby server is started (4). DRb.start_service is defined in 'drb/drb' (1). DRb.start_service is given the objects to be made public by dRuby, as well as the URI for the service. In this case, the SizedQueue object is made public at the URI "druby://localhost:9999". A dRuby server created by dRuby always has an object that encapsulates the service. This object is the "front" object.

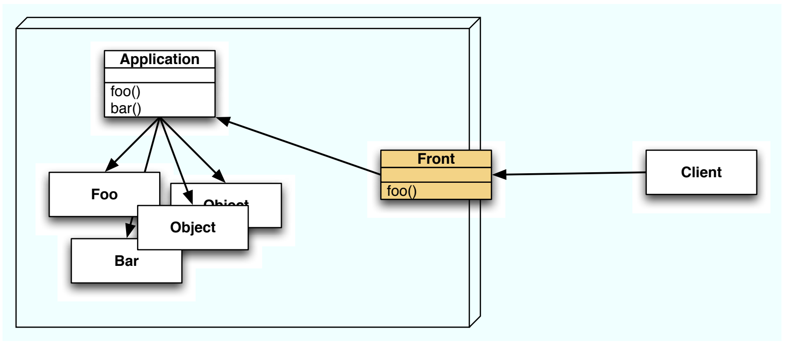

Fig. 2 front object and typical system

Finally, the main thread is stopped, without exiting, by calling sleep(5). Even though the main thread is stopped, the service continues to be available as it continues to run on threads in the background.

```
# producer
require 'drb/drb'

DRb.start_service                                          #(1)
queue = DRbObject.new_with_uri('druby://localhost:9999')     #(2)

100.times do |n|
  sleep(rand)
  queue.push(n)                                            #(3)
end
```

```
# consumer
require 'drb/drb'

DRb.start_service
queue = DRbObject.new_with_uri('druby://localhost:9999')

100.times do |n|
  sleep(rand)
  puts queue.pop                                          #(4)
end
```

The client-like consumer applications also call DRb.start_service (1). A call to DRb.start_service without any arguments indicates that the application has no front object. Note that applications that never export an object do not need to call DRb.start_service. Next, the reference object for the distributed queue is instantiated (2). DRbObjects are proxies referencing remote objects. A DRbObject instantiated with a URI references the object associated with that URI. Messages are then sent to the remote object (3).

To run the example, execute in separate terminals on the same host the scripts in this order: queue.rb, consumer.rb, producer.rb.

This simple example demonstrates how Ruby objects can be shared between processes in just a few lines of code. It is easy to write a distributed system in dRuby, just as it is easy to write applications in Ruby. Not only is dRuby suitable for writing distributed systems for prototyping or learning architectures, but such systems can also be implemented for use in production.

3.1.2 Implementation of RMI This section explains dRuby's implementation of RMI using code from the first version of dRuby[3]. The first version of dRuby is easy to understand, as it does not cover as much of the complexity as later versions. When a method is invoked on a DRbObject that is pass-by-reference, a message is transmitted to a remote dRuby server along with an identifier. The dRuby server uses the identifier to search for an object and invokes an appropriate method. Let's have a look at the implementation of DRbObject.

```
class DRbObject
  def initialize(obj, uri=nil)
    @uri = uri || DRb.uri
    @ref = obj.id if obj
  end

  def method_missing(msg_id, *a)
    succ, result = DRbConn.new(@uri).send_message(self, msg_id, *a)
    raise result if ! succ
    result
  end

  attr :ref
end
```

The only method defined in DRbObject is method_missing. In Ruby, method_missing is the method called when the receiver object receives an unknown method – i.e. a "missing method" is invoked. method_missing in DRbObject connects to the dRuby server, sends the ID of the corresponding object and the name of the invoked method and its arguments, and then returns the result.

The methods "push" and "pop" in the earlier script are examples of this. These methods are not defined in DRbObject, so method_missing is invoked, and the message is forwarded to the dRuby server specified by the URI given on initialization.

```
class DRbServer
  ...
  def proc
    ns = @soc.accept
    Thread.start(ns) do |s|
      begin
        begin
          # Ruby supports parallel assignment.
          # An assignemnt expression have one or more lvalues.
          ro, msg, argv = recv_request(s)
          if ro and ro.ref
            obj = ObjectSpace._id2ref(ro.ref)
```

[3] http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-list/15406

```
          else
            obj = DRb.front
          end
          result = obj.__send__(msg.intern, *argv)
          succ = true
        rescue
          result = $!
          succ = false
        end
        send_reply(s, succ, result)
      ensure
        s.close if s
      end
    end
  end
  ...
end
```

The pseudo code above describes the main behavior of the dRuby server. After "accept"ing a socket, a new thread is created and the object's identifier, method to be invoked and arguments are received. After finding the object corresponding to the identifier, the method is invoked on it. In most distributed object systems, it is common for a proxy defining the forwardable methods to be prepared for the class of each receiver. In dRuby, however, the usage of Ruby's method_missing method and runtime method lookup mechanism allows a single kind of proxy to act as the client stub for all classes. When the remote receiver (i.e. the original object) receives a message it cannot handle, it raises a NameError exception – the same behavior as a regular Ruby object.

Notice that a new thread is created for each individual message. Creating threads has a number of advantages, such as reducing the likelihood of exceptions by providing an independent execution context and avoiding network IO blocking. The greatest advantage for users, however, is that multiple RMI is permitted. This is the reason why the earlier producer-consumer problem program does not deadlock. When the pop invoked by the consumer causes the DRbServer to block, the producer is still able to push. Even when dRuby is in the middle of an RMI, it is still possible to RMI. As a result, callback style execution, iterators and recursive calls will work in dRuby, as well as the usage of thread synchronization mechanisms across processes. Because RMI processing is multithreaded, the user does have the responsibility to ensure thread safety.

3.2 Message format

This section describes the message format and the mechanism for object exchange.

dRuby uses TCP/IP for networking and the Marshal class library as its encoding mechanism. [4]

Marshal is a unique object serialization library, included as part of Ruby's core libraries. Starting from the supplied object, Marshal traces references one-by-one to serialize the entire graph of related objects. dRuby makes heavy usage of the

[4] The network and the message format can be customized.

Marshal library. In some systems, only objects with the "Serializable" property are considered to be serializable. In Ruby, however, all objects are initially considered to be serializable. When Ruby encounters an object whose serialization is meaningless, or when serialization proves to be impossible (such as File, Thread, Proc, etc.), an exception will be raised.

The request forwarded to remote object is composed of the following set of information (each component is an object serialized by Marshal).

- object identifier
- method string
- arguments

Let's have a look at the implementations of the first version. The latest version also sends length of each component, but it does not have any other essential difference.

```
def dump(obj, soc)
  begin
    str = Marshal::dump(obj)   # (1)
  rescue
    ro = DRbObject.new(obj)    # (2)
    str = Marshal::dump(ro)    # (3)
  end
  soc.write(str) if soc
  return str
end
```

This short snippet of code is one of the distinguishing parts of dRuby's implementation. First, the object passed as an argument is serialized using Marshal.dump(1). In the case that an exception is raised, the reference to the object (i.e. the DRbObject created (2) with the object) is serialized using Marshal.dump. (3) When Marshal.dump fails, this means that the target object was unserializable, or an unserializable object was referenced by the target object. In this case, dRuby does not allow the RMI to fail. Unserializable objects, or objects that cannot be passed as values, are instead passed as references to the object. This behavior is one of the tricks that dRuby uses to minimize the gap between dRuby and vanilla Ruby.

Let's see it in action. In this example, we will prepare a shared dictionary. One service registers services in the dictionary, while another service uses the services in the dictionary.

We'll deal with the dictionary service first. In distributed systems parlance, we might refer to it as the name server. Since we're just making a Hash public, it's very short.

```
# dict.rb
require 'drb/drb'

DRb.start_service('druby://localhost:23456', Hash.new)
sleep
```

Next is a log service. This is a simple service that records a time and a string. The SimpleLogger class defines the main logic. Running logger.rb will register a

SimpleLogger object and a description with the dictionary service, and then sleep. In this example, DRb.start_service is called without the argument. The logger service object is not explicitly exported. It happens implicitly when the logger is assigned to the hash table by the object passing mechanism described in section 2.3. Since the SimpleLogger references a File object (in this script, standard error output), it cannot be serialized with Marshal.dump. Consequently, it cannot be passed by value, and is passed by reference instead.

```
# logger.rb
require 'drb/drb'
require 'monitor'

DRb.start_service                                  # (1)
dict = DRbObject.new_with_uri('druby://localhost:23456')

class SimpleLogger
  include MonitorMixin

  def initialize(stream=$stderr)
    super()
    @stream = stream
  end

  def log(str)
    s = "#{Time.now}: #{str}"
    synchronize do
      @stream.puts(s)
    end
  end
end

logger = SimpleLogger.new
dict['logger'] = logger
dict['logger info'] = "SimpleLogger is here."
sleep
```

Finally, I will show how to use the log service. The script app.rb creates a reference to the dictionary service and retrieves the logger service using the logger's description string. After inspecting each object with the p method, we see that the info object is simply the string object "SimpleLogger is here.", while the logger object is revealed to be a DRbObject.

```
# app.rb
require 'drb/drb'

DRb.start_service
dict = DRbObject.new_with_uri('druby://localhost:23456')

info = dict['logger info']
logger = dict['logger']
```

```
p info      #=>  "SimpleLogger is here."
p logger    #=>  #<DRb::DRbObject:0x....>

logger.log("Hello, World.")
logger.log("Hello, Again.")
```

logger.log() is an RMI that generates log output. We should be able to check the output in the terminal running logger.rb.

In this chapter, I described two distinguishing features of dRuby's implementation with working examples: the method call implementation, and the implementation of the mechanism for selecting the method of object transmission and message format. Extracts from the first version of dRuby were used in the explanations, but the reader should be aware that the essence of the implementation remains unchanged in the current version.

4 Performance

In this chapter, I discuss results from testing dRuby's RMI performance.

The data produced by the following experiment indicates the maximum possible RMI rate between processes on the same machine. The results from this experiment should be considered to be for a best-case scenario, and not results for typical usage. The results should give a good reference for the degree of overhead in dRuby.

```
# count_test.rb
require 'drb/drb'

class Test
  def count(n)
    n.times do |x|
      yield(x)
    end
  end
end

DRb.start_service('druby://yourhost:32100', Test.new)
sleep
```

count_test.rb calls back caller's block using yield.

```
# count.rb
require 'drb/drb'

DRb.start_service(nil, nil)
ro = DRbObject.new_with_uri('druby://yourhost:32100')
ro.count(10000) {|x| x}
```

I measured the time taken for 10,000 remote method invocations in two different environments. The first case measures transmission between guest OS running on a virtual machine on a host OS, and the host OS on the same machine (Pentium4 3.0GHz). This combination is Ruby on Windows XP, and Ruby on a coLinux instance running as a guest OS within the Windows XP host OS.

```
% time ruby count.rb
real   0m11.250s
user   0m0.810s
sys    0m0.260s
```

For the following set of results, RMI is not used. Instead, regular method calls within the same process were tested.

```
% time ruby count.rb
real   0m0.044s
user   0m0.040s
sys    0m0.010s
```

The next two sets of results are from executing the tests on an iMac G5. The first set of results are with separate processes on the same OS, while the second set of results used regular method calls within a single process.

```
real    0m13.858s
user    0m6.517s
sys     0m1.032s

real    0m0.079s
user    0m0.031s
sys     0m0.012s
```

Between 700 - 900 remote method invocations per second were achieved. Whether this is sufficient for your application is for you to decide. Note that regular method calls within the same process (without RMI) were approximately 200 times faster. The frequency of RMI is likely to have a significant impact on application performance, and is an important point to consider when building real applications.

The last result set is from executing the tests with two machines (iMac G5 and iBook G4) connected by wireless LAN (IEEE 802.11b / 11Mbps).

```
real 0m47.010s
user 0m11.494s
sys 0m2.794s
```

5 Applications

In this section, I introduce a few applications which use dRuby. I also discuss a distributed coordination system based on dRuby, Rinda, and a practical example. While not discussed here, it is interesting to note that the Ruby on Rails debugger makes use of dRuby, as does the asynchronous processing associated with a number of Web applications.

5.1 Backend service of large scale Web application

The Hatena screenshot service is an example of a backend use in large scale Web applications. The Hatena screenshot service is a service reported by Tateno at

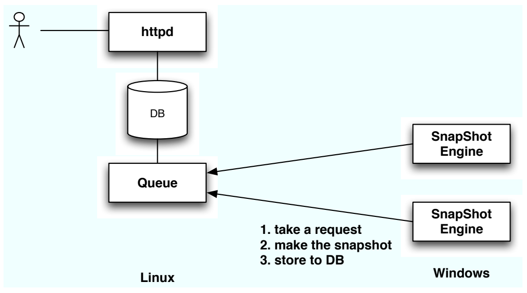

Fig. 3 hatena snapshot service

Ruby Kaigi 2006. The screenshot service provides thumbnails of, for example, registered URL screenshots to other Hatena services like blogs. The web front end is run on Linux, but the screenshot processing is implemented using Windows IE components, because this achieves better performance. The Linux front end uses dRuby to send objects containing a URL and return method to the screenshot processes running on Windows.

According to data presented at Ruby Kaigi 2006, this system handles about 120 SS/min (170,000 SS/day) using two backend machines.

5.2 In-Memory Database, or Persistent Process state

Next, I shall introduce the usage of dRuby as persistent memory, or as a persistent process state. Many problems in web applications relate to the need for both processes that deal with short-lived request-response cycles, and semantic processes.

One concrete example is CGI. CGI programs are invoked upon receiving a single request, and finish after returning a response. From the point of view of the user on their web browser, however, the application appears to have a much longer life-cycle. In order to make a series of small request-response cycles feel like a single long-lived application, each CGI instance must leave a “will” for the next generation before it dies.

The management of these “wills” (session management) is one of the major pain points in web application programming. Many factors need to be taken into account – the serialization of state, handling mutual exclusion in files or relational databases, and dealing with conflicts arising from multiple, simultaneous requests.

One approach is to minimize, or even eliminate, the “will” left for the next process by combining the short-lived front-end with a long-lived, persistent application.

RWiki is an interesting WikiWikiWeb implementation that applies such an architecture.

Meta-data such as the source of the Wiki page, the cache of the HTML output, links and update time stamps, are all maintained in memory on a long-lived server process. The short-lived CGI processes access this server via dRuby to retrieve Wiki pages requested by the user. One private RWiki server hosts approximately 20,000 pages in memory. In order to be able to rebuild the site after a reboot, the server constantly logs sufficient data on disk. These logs, however, are never referenced during normal execution.

The following example is an extremely small CGI script (4 steps), along with a simple "counter" server. Let's quickly review the mechanism of CGI. A CGI process retrieves a HTTP request from a CGI environment (typically a web server) via standard input (stdin) and environment variables, and then returns a response through standard output (stdout).

This CGI script invokes the comparatively long-lived counter server and passes it the environment variables and references to the standard output and input. This is the frontend.

```
#!/usr/local/bin/ruby

require 'drb/'drb'

DRb.start_service('druby://localhost:0')               # (1)
ro = DRbObject.new_with_uri('druby://localhost:12321') # (2)
ro.start(ENV.to_hash, $stdin, $stdout)                 # (3)
```

When the frontend invokes the remote object's start method (3), it effectively exports objects representing standard input and standard output, so it has to act as a dRuby "server". It starts the dRuby service on unspecified port of localhost (1). The OS will pick unused port. Next, the reference object for the counter server is instantiated (2). Finally, the frontend invokes the "start" method passing the environment variables by value as a hash and standard input and output by reference (3). "start" method is inherited from WEBrick::CGI.

```
require 'webrick/cgi'
require 'drb/drb'
require 'thread'

class SimpleCountCGI < WEBrick::CGI
  def initialize
    super
    @count = Queue.new
    @count.push(1)
  end

  def count
    value = @count.pop
  ensure
    @count.push(value + 1)
  end

  def do_GET(req, res)
    res['content-type'] = 'text/plain'
    res.body = count.to_s
  end
end

DRb.start_service('druby://localhost:12321', SimpleCountCGI.new)
sleep
```

6 Rinda and Linda

Linda is a simple coordination model based on a virtual shared memory organized into tuples and tuple spaces that enables coordinate execution of multiple tasks. It is a very attractive model that can manage various parallel programming challenges, even though it is simple. Because of this, the Linda model has been incorporated into many languages. C-Linda, JavaSpace and Rinda are typical examples.

C-Linda enhances the base language C by implementing Linda operations using a combination of preprocessing and a runtime support library. JavaSpace provides a collection of Java classes that implement the Linda model. The implementations of Ruby's tuple space is provided by the library Rinda which makes use of dRuby. C-Linda provides a single tuple space, while JavaSpaces and Rinda allow for multiple tuple spaces.

Rinda provides the basic Linda operations: out, in, inp, rd, rdp. Eval is not provided, but it can be replaced in some situations by the use of Ruby threads.

The latest version of Rinda follows JavaSpaces's naming conventions for the operations:

write (out) Add a tuple to tuple space.
take (in) Delete a matching tuple from the tuple space, and return the deleted tuple. If a matching tuple does not exist, block. “matching” is discussed below.
read (rd) Return copy of a matching tuple. If matching tuple does not exist, block.

Take and read accept timeouts. If the timeout is set to zero, they behave similar to inp and rdp. Apart from these basic operations, “read_all” is also available to read all tuples matching given patterns. “read_all” is particularly useful for debugging.

Tuples and patterns are represented by Ruby arrays. Patterns for matching tuple are more general than C-Linda and include ones natural to Ruby. Wild cards (nil indicates a wild card in Rinda), classes, ranges and regexps can all be specified. Rinda, in effect, can be used as a sort of query language.

A time limit can be set on a tuple, although this function is still experimental. Both times in seconds and life time update objects (called renewers in Rinda) can be specified for the time limit. A renewer will be asked whether or not to renew the life time of a tuple. Rinda allows a renewer to be a dRuby's reference. Using this feature, we can, for instance, provide for the expiration of some tuples in the event that a tuple creator is closed abnormally[8],[10].

6.1 Dining philosophers

In terms of Rinda, I explain to you with actual code.

```
require 'rinda/tuplespace'                        # (1)

class Phil
  def initialize(ts, num, size)                   # (2)
    @ts = ts
    @left = num
    @right = (@left + 1) % size
    @status = ' '
  end
  attr_reader :status

  def think
    @status = 'T'
    sleep(rand)
    @status = '_'
  end

  def eat
    @status = 'E'
    sleep(rand)
    @status = '.'
  end

  def main_loop                                  # (3)
    while true
      think
      @ts.take([:room_ticket])
      @ts.take([:chopstick, @left])
      @ts.take([:chopstick, @right])
      eat
      @ts.write([:chopstick, @left])
      @ts.write([:chopstick, @right])
      @ts.write([:room_ticket])
    end
  end
end

ts = Rinda::TupleSpace.new                       # (4)
size = 10
phil = []
size.times do |n|
  phil[n] = Phil.new(ts, n, size)
  Thread.start(n) do |x|                         # (5)
    phil[x].main_loop
  end
  ts.write([:chopstick, n])
end

(size - 1).times do
  ts.write([:room_ticket])
end

while true                                       # (6)
  sleep 0.3
  puts phil.collect {|x| x.status}.join(" ")
end
```

The well-known “dining philosophers” problem is introduced here. This program uses two types of tuples. One is a chopstick and the another is a room_ticket. A chopstick tuple has two elements. The first element is the symbol[5] “:chopstick”, and the second element is an integer indicating the chopstick’s number. room_ticket tuples limit the number of philosophers let into the dining room—a philosopher must have a room_ticket to enter the room. The tuple contains just a symbol “:room_ticket”.

Phil class models a philosopher. The object has instance variables indicating status(2). These variables are required to monitor a philosopher’s status. The main_loop method in the Phil class is an infinite loop simulating a philosopher’s actions(3).

After executing think(), a philosopher gets a ticket that permits entry to the dining room and then picks up two chopsticks, first the one to the left and then the one to the right. Once all items are ready, the philosopher executes eat(). When finished dining, the right chopstick, left chopstick and the room ticket are returned to the tuple space, and the philosopher again executes think() to ponder the Way of Ruby.

The main program first creates a tuple space (4) and then creates the philosophers, invoking the main loop method of each by subthread(5). These operations are similar to the eval() operation in C-Linda. As its is creating philosophers, the main program also creates chopsticks (one per philosopher, but keep in mind that two are needed to eat). Next the main program creates a number of room tickets equal to one less than the number of philosophers. The last infinite loop monitors the philosophers every 0.3 seconds(6). The loop displays the status of each philosopher: thinking, dining, or holding chopsticks.

6.2 Tuple and Pattern

As mentioned before, a tuple or pattern is represented as a Ruby Array. All Ruby objects, including dRuby’s reference object, can be used as an element of these Arrays. Here are some examples of tuples:

```
[:chopstick, 2]
[:room_ticket]
['abc', 2, 5]
[:matrix, 1.6, 3.14]
['family', 'is-sister', 'Carolyn', 'Elinor']
```

The matching rules for tuples and patterns are a little unusual. nil is interpreted as a wild card which can match any object. Each non-nil pattern element is compared with the corresponding tuple element by using the relationship operator “===”.

“===” is a special equality comparison, similar to “==”. In certain situations, however, it is more general. For example, when a pattern element is a Regexp, “===” is nothing but pattern matching of strings; while if the pattern element is a Range, “===” indicates whether or not the corresponding tuple element is within the range limits. When a Class is specified as the pattern element, “===” behaves as kind_of?(), so that patterns which specify a class can be described as

[5] Symbol is a Ruby Class.

well. kind_of?() returns true if Class is the class of object, or if Class is one of the superclasses of object.

The followings are some examples of patterns.

```
[/^A/, nil, nil] (1)
[:matrix, Numeric, Numeric] (2)
['family', 'is-sister', 'Carolyn', nil] (3)
[nil, 'age', (0..18)] (4)
```

1. Matches a tuple with three elements and the first element must be a string that starts with “A”.
2. Matches a tuple which has the symbol“matrix” as its first element, and the second and the third elements match the class Numeric.
3. Matches a tuple representing “Carolyn’s sister”.
4. Matches a tuple representing an age between 0 and 18.

Let’s test these patterns by using a shared TupleSpace server and the interactive environment irb. First, we start the server with these four lines:

```
require 'rinda/tuplespace'
ts = Rinda::TupleSpace.new
DRb.start_service('druby://localhost:12121', ts)
sleep
```

Next, we check patterns using an interactive Ruby session.

```
% irb --simple-prompt -r rinda/rinda
>> DRb.start_service
>> ts = DRbObject.new_with_uri('druby://localhost:12121')
>> ts.write(['seki', 'age', 20])
>> ts.write(['sougo', 'age', 18])
>> ts.write(['leonard', 'age', 18])
>> ts.read_all([nil, 'age', 0..19])      # using Range object (0..19)
=> [["sougo", "age", 18], ["leonard", "age", 18]]
>> ts.read_all([/^s/, 'age', Numeric])   # using Regexp object (/^s/)
                                         # and Numeric class
=> [["seki", "age", 20], ["sougo", "age", 18]]
>> exit
```

You can see that patterns offer a Ruby-like flexibility. You might also wish to use tuple spaces as a simple data base. Keep in mind that Rinda conducts a linear search to match patterns and tuples, so you have to be careful when dealing with a large number of tuples.

6.3 Special-case optimization

The current Rinda implementation uses a special case optimization in order to perform high-speed searches. Experience indicates that tuples of the following kind are often used by applications:

```
[:screenshot, 12345, "http://www.druby.org"]
[:result_screenshot, 12345, true]
[:prime, 10]
```

That is to say, it is common to use tuples composed of a symbol acting as a message "type" and some arguments. When using these kinds of tuples, take and read often use patterns like:

```
[:screenshot, nil, String]
[:result_screenshot, 12345, nil]
[:prime, Numeric]
```

Given this situation, you can probably obtain better performance by using the first symbol element as a key that maps to a collection of tuples whose first element is that symbol, and then focusing the search on that collection. We cannot use this in general, because not all objects can be used as a key, but a symbol is appropriate and a symbol is as easy to read as a string. When nil is used as first element of a pattern, all collections are searched.

6.4 Application of Rinda

Buzztter is a Web service that processes Twitter sentences. Twitter is a SNS[6] specialized for short sentences. Buzztter collects sentences posted into Twitter, and figures out which words are used more often than usual. By doing so, Buzztter understands the overall trend of words usage at a given moment in Twitter.

Buzztter is composed of several subsystems. One of these is a distributed crawler sub-system that uses Twitter's API (HTTP based) to collect sentences. The crawler subsystem has multiple fetchers that fetch information from Twitter and importers to make it persistent. Rinda and dRuby are used as mediators between the fetchers and the importers. For reference, the following indicates the data rates handled by Buzztter (as of Nov 3, 2007)

– 72MB per day
– 125000 transaction per day

6.5 Persistent Rinda

A Rinda release with persistent tuple spaces was announced at Ruby Kaigi 2007. Normally the contents of a TupleSpace instance disappear once processes using it finish. The contents of a persistent TupleSpace instance, however, are available after reboot. In order to be able to rebuild the TupleSpace after a reboot, the persistent TupleSpace constantly logs all activity (write, take) on disk.

7 Conclusion

I have discussed dRuby's concept, design, implementations and provided examples of practical usage. In addition, I have discussed the Rinda implementation of Linda's tuple space, based on dRuby. Both dRuby and Rinda are designed to be simple systems in order for Ruby programmers to feel comfortable with them. Hence, they are very appropriate for prototyping a distributed system. The practical examples discussed in this article, however, illustrate that they are not just "toys" of limited use. Those examples clearly demonstrate that dRuby and Rinda are robust infrastructure for building practical applications.

[6] Social Networking Service

Acknowledgements

I'd like to thank Hisashi Morita, Leonard Chin, Mayumi Morinaga, Sougo Tsuboi and Takashi Egawa for translating and reviewing.

References

1. Hatena::screenshot. http://screenshot.hatena.ne.jp/.
2. Kent Beck. *Smalltalk: best practice patterns.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
3. Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.
4. Nicholas Carriero and David Gelernter. How to write parallel programs: a guide to the perplexed. pages 52–86, 1995.
5. David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
6. Bo Leuf and Ward Cunningham. *The Wiki way: quick collaboration on the Web.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
7. Scott Oaks and Henry Wong. *Jini in a nutshell: a desktop quick reference.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.
8. Masatoshi Seki. *Distributed and Web Programming with dRuby.* Ohmsha, 2005.
9. Masatoshi SEKI. druby again. http://www.druby.org/dRubyAgain.pdf, 2006. Presentation at RubyKaigi 2006, Tokyo Japan.
10. Masatoshi SEKI. Rinda: Answering the rubyconf, rubykaigi. http://www.druby.org/RK07.pdf, 2007. Presentation at RubyKaigi 2007, Tokyo Japan.
11. Robert J. Sheehan. Teaching operating systems with ruby. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 38–42, New York, NY, USA, 2007. ACM.
12. Youji Shidara. Inside buzztter. http://www.slideshare.net/dara/buzztter, 11 2007.
13. Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
14. David Thomas and Andrew Hunt. *Programming Ruby: the pragmatic programmer's guide.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.