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 loved 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 from the enterprise. 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, performances and examples of common practices. An English translation of the author's Japanese dRuby book is currently being planned by the Pragmatic Bookshelf.

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 sketching out distributed systems, this article will also demonstrate that dRuby and Rinda are building a reputation for being suitable as components of the infrastructure 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.

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 thread

Ruby is so-called, categorized as a dynamic object-oriented language. Everything is composed of objects, and there is no type of variables. Unification of methods is all made at execution time. Furthermore, Ruby has rich reflection functionality and allows to use metaprogramming. Ruby is a mysterious language, as if the creator made tricks on Ruby. We do programming as Ruby leads us, and without notice we feel like almost touching on the essence of object-oriented programming.

dRuby is a distributed object environment for making Ruby running on it. dRuby extends Rubys method calls across the network and enables to call object methods from other process/machines. Another is that Rinda incorporates Lindas implementations within itself; Linda is a glue language of a distributed coordination system based on dRuby, so it provides common tuple spaces. This article introduces not only dRubys concept and its design policy but also its implementations and practical usage. This article will be discussed as follows.

- the way of dRuby - dRuby's overview and its design policy
- Implementation 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

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

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

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

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

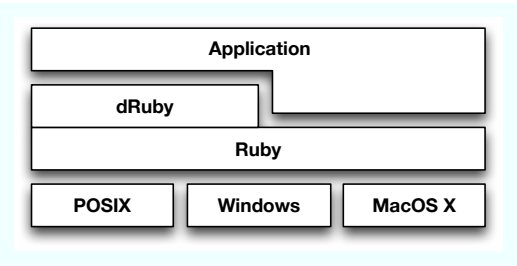

Fig. 1 dRuby application software structure

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.

As a result, dRuby extends the Ruby interpreter across other processes and other machines, both in a physical sense and a temporal sense.

dRuby has the following characteristics.

- exclusively for Ruby
- written purely in Ruby
- specifications, such as IDL, not 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
- no distinction between server and client

dRuby is a distributed object system exclusively for Ruby. Only scripts written in Ruby can be handled by dRuby, however, dRuby can run on any machine that runs Ruby, irrespective of operating system. That is, even when running on different platforms, as log as Ruby runs, the platforms will be able to exchange objects, and also use methods on each others objects.

dRuby is written completely in Ruby, not a special extension written in C. Thanks to Ruby's excellent thread, socket and marshalling 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 it dRuby is available wherever Ruby is installed.

2.2 Compatibility with Ruby

dRuby pays special attention to maintaining compatibility with Ruby scripts. 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 Ruby's unique benefits of Ruby are available for the programmer to enjoy.

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

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 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 minimises the need for programmers to care about the difference between types of object exchange, while also striving to be reasonably efficient. In dRuby,

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.

Although this rule may not always be correct, in most situations it will work. Here, I would like to briefly discuss this rule. 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 has been provided, however, there is a risk that circular references will give rise to objects that never get garbage collected. 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 as it is, so dRuby is not just a standard Ruby-like interface of RMI. dRuby would rather co-exist with XML-RPCSOAPCORBA e.g. In fact, some use http as interface for external network and where dRuby is incorporated in their internal systems at the backend.

3 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

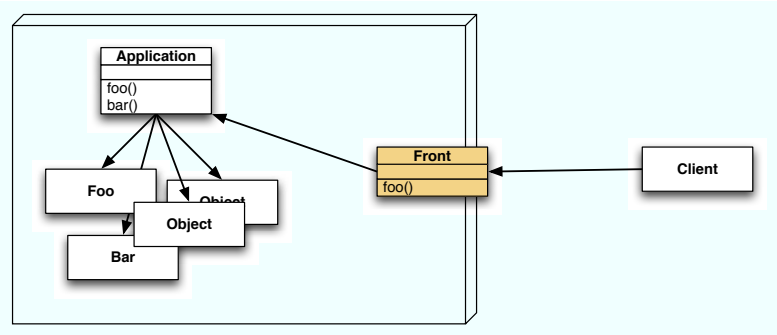

Fig. 2 front object and typical system

code, I will describe in detail how basic RMI is and the mechanism of object transmission.

3.1 Basic RMI

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

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

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

queue = SizedQueue.new(10)                           # (2)
DRb.start_service('druby://localhost:9999', queue)   # (3)
sleep                                                # (4)
```

First, I shall explain the shared queue server. Applications using dRuby must start by loading 'drb/drb' (1). Next, a SizedQueue object (3) with a limited number of buffer elements is instantiated. Then the DRb services is started (3). 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". Any systems created by dRuby always have an object that indicates the system entrance. The object is called as a front object.

Finally, the service is stopped, without exiting, by calling sleep(4). 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 producer and 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).

This example can be executed by preparing three terminals, and executing the scripts in order in separate terminals. No other special set-up is required.

```
% ruby queue.rb
```

```
% ruby consumer.rb
```

```
% ruby 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 The following explains dRuby's implementations in RMI by referring from the first version of dRuby[2]. As the first code does not describe detail, it is not difficult to find the essence. When a message is sent to DRbObject - a remote object referring a reference object- the message is transmitted to a remote dRuby server along with a receiver identifier. dRuby searches an object from the receiver identifier and invokes a method. Let's have a look at DRbObject in the first version of dRuby.

```
class DRbObject
  def initialize(obj, uri=nil)
```

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

```
    @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
```

In the DRbObject, method_missing is just defined. The method_missing is invoked only when receivers receive unmanageable messages. The method_missing of the DRbObject establishes connections with dRuby server where objects are present. The method sends the receiver ID, the method name and its arguments, and returns the result. That is to say, a method undefined in the DRbObject is all transmitted by remote. push and pop in the script shows what I just discussed here. They are not defined in DRbObject so that method_missing is called, and it transmits messages to an intended dRuby server specified in URI.

```
class DRbServer
  ...
  def proc
    ns = @soc.accept
    Thread.start(ns) do |s|
      begin
        begin
          ro, msg, argv = recv_request(s)
          if ro and ro.ref
            obj = ObjectSpace._id2ref(ro.ref)
          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
```

pseudo code above describes main process of dRuby server. After accepting socket, it creates threads and receives an object identifier, messages and argu-

ments. After searching a target objects as per the identifier, send the received messages. In most of the distributed object systems, proxy is created for each receiver class, where a list of transferable messages is defined. However, in the dRuby, the mechanism of method_missing and the message search at the execution time enables one proxy to function as a client stab for all classes. If dRuby receives message unmanageable by remote receivers (original objects), similar to Ruby's standard operations executed to objects, NameError exception is raised.

Another point is that threads are created as per individual message. This helps enhance dRuby against exceptions by preparing independent execution context. For instance, avoid blocks of network IO e.g. The best advantage for end-users is to allow multiple RMI. The reason why the previous program of producer-consumer problem did not end up in dead lock is just because of the following tips. While DRbServer process has been blocked due to pop by consumer, producer can execute push, which means dRuby allows executing RMI while executing other RMI. This mechanism enables to execute call back-like processes, iterators or recursive calls, and synchronize threads among processes.

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. [3]

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 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.

- receiver identifier
- message string
- arguments

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

```
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
```

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

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 to the dictionary service, and then sleep. 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
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 explain the service user. 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, the info object is simple 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 usable 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. The first version of dRuby is the most suitable for reviewing the implementation.

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 number of RMI 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.

```
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
```

```
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 is 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.

5 Applications

There are a few applications that made implementations by using dRuby. As is well known, in the Ruby on Rails debugger is implemented and in asynchronous process of Web applications where dRuby are often implemented. In this chapter, I introduce a few applications which made implementations by using dRuby. On top of that, I also discuss a distributed coordination system based on dRuby, Rinda and a practical example.

5.1 Backend service of large scale Web application

Here, I introduce Hatena screenshot service as an example of a backend use in large scale Web applications. Hatena screenshot service is a service reported by Tateno in Ruby Kaigi 2006. The screenshot service is to display thumbnails e.g. of registered URL screenshots to other Hatena service like blogs. Web front end is configured on Linux, but screenshot is implemented based on Windows IE components, because taking the screenshot under Window's environment achieves better performance at high speed. The screenshot is executed as asynchronous batch process from a front-end. Processes running on Windows receive objects of both a URL from processes on Linux via dRuby and a return method, and execute the screenshot.

According to data as of 2006 that RubyKaigi present, the handling scale in this system was likely 120SS/min, 170,000SS/day by parallelism with 2 machines.

5.2 On-Memory Database, or Persistent Process

Next, I shall introduce the usage of dRuby as persistent memory, or as a persistent process. 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" to 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 minimise, 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. 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. By replacing the counter example with another example, a CGI application that doesn't use the "will" model can easily be written.

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

require 'drb/'drb'

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

```
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

Finally, I introduce Linda's implementations based on dRuby, Rinda and its practical use case. Linda is a concept of a glue language in the distributed coordination

system. A simple model of tuples and tuple spaces enables to coordinate multiple tasks. That is to say, it is very attracting model that can manage various situations caused due to a parallel programming, even though it is a simple. Because of this reason behind, many languages incorporate the tuple spaces of its own. C-Linda, JavaSpace and Rinda are typical examples of the implementation seen from the following.

C-Linda enhances a base languageCand adds Linda's operations, so these are achieved by executing preprocessors. Regarding the implementations of JavaSpace, the tuple space is implemented by using Java. The implementations of Ruby's tuple space are achieved in Rinda. That is, Rinda implemented Linda's tuple and tuple space model in Ruby.

In the case of C-Linda, operable tuple spaces are implicitly limited to one tuple space, so that the tuple space does not have to be specified. In another case of Rinda, as tuple space and objects are communicated by message, applications have to decide which tuple spaces will use.

In the case of Rinda, in the Linda's basic operations, out, in, inp, rd, rdp except for eval are available, but that can be substituted for Ruby' threads.

Latest version of Rinda changed the basic operation method names to that like JavaSpace.

write Add tuples to tuple space. (out)
take Delete matching tuples from the tuple space, and return the deleted tuples. If a matching tuples do not exist, block them. (in)
read Return copy of matching tuples. If matching tuples do not exist, block them. (rd)

Take and read is used to set timeout. If the timeout sets to zero, they behave similar to inp, rdp. Apart from these basic operations, read_all is also available to read all tuples matching to patterns. The read_all appears to be useful for a debug use.

The tuple and patterns are expressed by Array of Ruby. Regulations of matching tuple patterns are expanded to Ruby-like regulations, so that not only wild card(Wild card in Rinda means nil.) but also classes, further more Range and Regexp can be specified. Rinda can be handled like a sort of query language.

Furthermore, time limit can set in the tuples, though this function is still under experiment. Also, numbers indicating seconds and time line update objects (It is called renewer in Rinda.) can be specified in the time limit. Whether to renew the time line or not is enquired to the objects. Rinda can also give dRuby's reference as renewer.. For instance, a tuple creator is closed abnormally. After some time, tuples to turn to out of time limit can be provided.

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
```

A well- known dining philosophers is introduced here. This program uses two types of tuples. One is chopstick, and another is room ticket. Chopstick has a tuple with two elements. The first element is symbol :chopstick, and the second element is an integer indicating the chopstick's number. room_ticket is a ticket that limits the number of philosophers to let in the room. The element contains just a symble :room_ticket.

Phil class indicates a philosopher. The Phil object is generated along with tuple spaces, numbers, and number of tables. The object has instance variables indicating status(2). These variables are required to monitor philosopher's status. Compared to C-Linda, Rinda has to pass target tuple spaces. main_loop method in the Phil class is an infinite loop indicating philosopher's actions(3). After executing think(), the main_loop gets room_ticket for dining, and get a left chopstick and following right chopstick. Once all items are ready, execute eat(). When the dining finishes, return the left and right chopsticks and the room_ticket to the tuple space, and again return to think().

In the main program, firstly create tuple spaces (3), and create philosophers, and then invoke the main_loop method by subthread(5). These operations are similar to eval() operation in C-Linda. Chopsticks corresponding to number of people and room_ticket of 1 lower number are written to the tuple spaces.

The last infinite loop is a group to monitor philosophers every 0.3 seconds(6). The loop indicates their actions whether they are thinking, dining, or holding chopsticks e.g. at that moment.

6.2 Tuple and Pattern

Here, I explain tuples and patterns and pattern matching in Rinda. As I mentioned before, those tuples and the patterns are expressed in Array. There are characteristics in elements. For the elements, all Ruby objects including dRuby's reference can be specified.

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

Similarly for patterns, all Ruby's objects are available as elements. In terms of pattern matching, its regulations are a little strange. nil is interpreted as wild card which can match any objects and each element is compared by ===case equals.

Ruby has a case expression and the case expression is a branch just like c switch. ===case equals is special equality comparisons. === is basically similar to ==, however, in a certain class, it behaves like patterns. For example, Regexp is nothing but a pattern matching of strings, and Range identifies whether values are within the limited range or not. When a Class is specified as pattern elements, it has come to be justified by kind_of(), so that patterns with class-specified can be described we all.

The followings are given sample examples of the patterns.

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

1. A tuple made of three elements, and the first element starts with "A"-string
2. A tuple arranged by that the first element is symbol "matrix" and the second and the third element are a numeric class.
3. Tuple of Carolyn sister
4. Tuple aged from 0 to 18

Let's check that patterns are matching by using common TupleSpace server and interactive environment irb.

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

This 4 lines are the script of common TupleSpace.

```
% irb --simple-prompt -r rinda/rinda
>> DRb.start_service
>> ro = DRbObject.new_with_uri('druby://localhost:12121')
>> ts = Rinda::TupleSpaceProxy.new(ro)
>> ts.write(['seki', 'age', 20])
>> ts.write(['sougo', 'age', 18])
>> ts.write(['leonard', 'age', 18])
>> ts.read_all([nil, 'age', 0..19])
=> [["sougo", "age", 18], ["leonard", "age", 18]]
>> ts.read_all([/^s/, 'age', nil])
=> [["seki", "age", 20], ["sougo", "age", 18]]
>> exit
```

You can see that Ruby-like flexible patterns are available. You might also consider that tuple spaces can be used as a simple data base. In this point, you have to be careful in dealing with a large number of tuples, according to the nature of API with the pattern-flexible, Rinda conducts a liner search.

6.3 Unfair optimization

Latest Rinda has already implemented unfair optimization in order to perform searchings at a high speed by using general applications. Only when the first element is a symbol, store tuples to an own collection. As per my experience, the following type of tuples is often used by application side.

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

That is to say, it is tuples composed of a type of message and some arguments. In the case of take or read, the following patterns is applicable.

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

This is nothing but a pattern which takes any one from a certain message type tuples. Considering this situation, you can probably expect high performance by storing the first element as a key into own collection and focusing on search targets. On the other hand, there is another problem whether or not any objects can be used as a key to achieve a high performance. In the case of Rinda's pattern, the use of===case match comes to unsuitable for String and Integer as a key. However, Symbol is still appropriate as a key because ===case match of Symbol has a Symbol class and its values only. Furthermore, Symbol is as easy to read as String.

Let's summarize the unfair optimization here. In the latest Rinda, when the first element in the tuple is a Symbol, Rinda executes the unfair optimization, and performance on take/read is improved. Similarly for take/read, when a pattern search is that the first element is a Symbol, the searching performance comes faster than usual.

6.4 Application of Rinda

Here, I introduce a practical use of Rinda. Buzztter is a Web service interpreting Twitter sentences. The Twitter is SNS specialized on a short sentence. Buzztter collects posted sentences into Twitter, and interpret the sentences, and figure out words more often used than usual. By doing so, Buzztter understands overall trends of words of that moment in the Twitter.

Buzztter composes of several subsystems, in which a distributed crawler subsystem; a subsystem collects sentences by using Twitter API(HTTP); uses Rinda. The crawler subsystem is made of multiple fetchers that is to fetch information from Twitter and importers that is to make it persistent.Rinda and dRuby is a mediator between the fetchers and the importers. For your reference, the following is the data to be handled by Buzztter (as of Nov 3, 2007)

- 125000 case per day
- 72MB per day

6.5 Rinda Update

Lastly, I discuss Rinda's latest trends. Last year, in RubyKaigi 2007, persistent TupleSpace release was announced. Information of Rinda::TupleSpace disappears once processes finish. The persistent TupleSpace recovers a straight tuple space at the time of invoking processes again. During the execution, in order to be ready for re-invocation, the persistent TupleSpace keeps logging. At the time of invoking again, referring log information, the TupleSpace rebuilds the processes. While executing, the TupleSpace just keep logging, but it does mean to read contents in the storage.

7 Conclusion

I discussed dRuby's design policy and the implementations along with its concept, and introduced the practical usage. In addition to that, I discussed implementations of TupleSpace developed based on dRuby and Rinda. Both dRuby and Rinda are designed as a simple system in order for Ruby programmers to feel comfortable to know more of it. Hence, this is most appropriate to sketch the distributed system. However, practical examples discussed in this article are not for a toy discussion for sketching purpose. Those examples well demonstrated that dRuby and Rinda are practically available as infrastructure to build practical applications.

Acknowledgements

I'd like to thank Hisashi Morita, Leonard Chin, Mayumi Morinag, 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*. Ohm-sha, 2005.
9. Masatoshi SEKI. druby again. http://www.druby.org/dRubyAgain.pdf, 2007. 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 buztter. 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.