

Distributed Ruby

Mark Volkmann

Object Computing, Inc.

9/27/2005

Distributed Ruby

- Two forms of creating distributed applications with Ruby
- 1) dRuby or drb
 - remote object communication similar to Java RMI
- 2) Rinda
 - a tuplespace implementation

dRuby Overview

- Allows objects in one process to invoke methods on objects in another process
 - remote method invocation with “servers” and “clients”
 - processes can be on different hosts
- Doesn't use or interact with other distributed architectures
 - such as .NET, RMI and CORBA
- No server lookup provided
 - nothing like CORBA naming or trader service
 - clients must know host and port of “servers” they wish to use
- Both sides must have source code for classes of all objects passed
 - not like Java RMI which passes bytecode for classes at runtime

dRuby Overview (Cont'd)

- One host/port per “initial object”
 - a.k.a. “front object” or “server”
 - but those objects can have methods that return remote references to other objects
- Objects passed to or returned from server methods can be passed
 - **by value**
 - makes a copy so changes made by server aren't seen by client
 - serialized using `Marshal` module
 - **by reference**
 - remotely accessible
 - faster when passing large objects; avoids marshalling them
 - uses `DRbObject` objects (more on this later)
- **Security**
 - provided by IP-address-based ACLs
 - covered later

Obtaining dRuby

- Core classes are provided with Ruby
- Full package
 - download from
<http://www2a.biglobe.ne.jp/~seki/ruby/druby.en.html>
 - see “Download” link
 - also contains Rinda source files

Currently there are only a couple of .rb files in this that aren't in the Ruby distribution ...
`invokemethod16.rb` and `udp.rb`

Why Access Remote Objects?

- To access services that do not provide a code download
- To offload processing to another host
 - creating a distributed application
- To control access to a shared resource
 - such as a database
- And others ...

Writing a Server

```
class MyServer
  def get_greeting(name)
    return "Hello #{name}!"
  end
end

require 'drb'

# Set to nil to use localhost and allow drb to pick port.
uri = "druby://host:port"

initial_object = MyServer.new
DRb.start_service(uri, initial_object)
puts "URI is #{DRb.uri}" # useful when uri is nil
DRb.thread.join # don't exit so requests can be processed
```

Ctrl-c to exit under Unix;
Ctrl-Break under Windows

Writing a Client

```
require 'drb'
```

```
DRb.start_service ←
```

only needed if the client returns
object references to the server;
allows the client to act as a server
when methods on those objects are invoked

```
uri = 'druby://host:port'
```

```
proxy = DRbObject.new(nil, uri)
```

```
proxy = DRbObject.new_with_uri(uri) # same as previous line
```

local object for which a proxy is needed;
nil when creating a proxy for a given uri

```
puts proxy.get_greeting('Mark') # outputs "Hello Mark!"
```


Parameter & Return Value Marshaling

- Default behavior is pass-by-value
 - uses built-in `Marshal` module
 - marshals most kinds of objects along with all objects reachable from them
 - implemented in C; very fast
 - some kinds of objects cannot be marshaled so are always passed by reference
 - `Binding`, `Proc`, `IO` and singleton objects
- Can pass and return remote references
 - represented by `DRbObject` objects

Parameter & Return Value Marshaling (Cont'd)

- Ways to prevent marshaling and force pass-by-reference

- approach #1: include DRbUndumped in class

```
class Car
  include DRbUndumped
  ...
end
```

all objects created from this class
will be passed by reference
using DRbObject

- approach #2: extend an object with DRbUndumped

```
myCar = Car.new ...
myCar.extend DRbUndumped
```

if passed to a remote method, a
DRbObject will actually be passed

- approach #3: explicitly pass a DRbObject

```
myCar = Car.new ...
objToPass = DRbObject.new(myCar)
```

- Can customize what and how attributes are marshaled
 - see pickaxe p. 415

Marshal Example

```
class Car
  attr_accessor :make, :model,
               :year, :owner

  def to_s
    "#{year} #{make} #{model} " +
    "owned by #{owner}"
  end
end

class Person
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def to_s
    name
  end
end
```

```
me = Person.new("Mark Volkmann")
car = Car.new
car.make = 'BMW'
car.model = 'Z3'
car.year = 2001
car.owner = me
```

write new file
if already exists

```
File.open("car.rbm", "w+") do |f|
  Marshal.dump(car, f)
end
```

```
File.open("car.rbm") do |f|
  unmarshaled_car = Marshal.load(f)
  puts unmarshaled_car
end
```

outputs "2001 BMW Z3 owned by Mark Volkmann"

Threading

- Each remote method invocation is handled by a new server thread
 - so multiple clients don't block each other
- Consider handling of concurrent requests when implementing server methods
 - can use `Monitor`, `Mutex` or `Sync` libraries to provide synchronized access to data

Safety

- Tainted data
 - any data not create or modified by local Ruby code
- Value `$SAFE` global variable controls safety restrictions enforced by the Ruby interpreter
 - 0 (default)
 - no checking for use of tainted data
 - ≥ 1
 - tainted data cannot be used by some methods such as `eval`
 - ≥ 2
 - disallows loading Ruby code from “globally writable locations”
 - means all users on the system have write permission
 - ≥ 3
 - newly created objects are marked as tainted
 - ≥ 4
 - disallows modification of non-tainted objects

Set `$SAFE` to 1 or higher in server code to disable use of `Kernel.eval` and `Object.instance_eval` so clients can't execute arbitrary Ruby code in server. An example of this is in the comments at the top of `drb/drb.rb`.

can't get this to work

can't get this to work

Security with Access Control Lists

- **ACL** class is part of dRuby
- Constructor takes an array of strings
 - can create with %w
- **Example**

```
acl = ACL.new %w(  
  deny all  
  allow localhost  
  allow 130.76.110.*)  
DRb.install_acl acl
```

allows access from same host
or any host whose IP address
starts with 130.76.110

get a `DRb::DRbConnError`
in client when access is blocked

dRuby Components

- Three main components
 - remote method call marshaller/unmarshaller
 - uses the `Marshal` module
 - transport protocol
 - opens network connections and sends messages across them
 - manages marshalling with `DRb::DRbMessage`
 - protocol is selected by the scheme at the front of URIs
 - the scheme `druby:` uses `DRb::DRbTCPSocket` which uses TCP/IP sockets
 - a sample using HTTP is included
 - id to object mapper
 - remote references map to objects using host, port and object id
 - by default, maps dRuby ids to objects using `DRb::DRbIdConv`
 - this uses the `ObjectSpace` ids assigned to objects
 - only valid for life of process that created the objects
- Can override each component
 - to provide different behavior

can configure to use **SSH** or **SSL**;
see [RubyGarden DrbTutorial](#)

dRuby Documentation

- **Comments at top of `drb/drbrb.rb`**
 - Masatoshi Seki
- **Intro to DRb**
 - Chad Fowler
 - <http://www.chadfowler.com/ruby/drbrb.html>
- **DrbTutorial**
 - <http://www.rubygarden.org/ruby?DrbTutorial>
 - describes configuration to use SSH and SSL
- **Where Ruby Really Sparkles**
 - Dave Thomas
 - http://www.linux-mag.com/2002-09/ruby_01.html

Tuplespace Overview

- **Began with “Linda”**
 - see “About Linda” at <http://www-users.cs.york.ac.uk/~aw/pylinda/about.html>
- **Terminology**
 - **Tuple**
 - an ordered collections of values (objects in Ruby)
 - **Tuplespace**
 - shared memory that holds tuples
 - sometimes referred to as a whiteboard or bulletin board
 - **Template**
 - a tuple where some values are names of data types or patterns
 - used to match tuples
- **Basic operations**
 - add a tuple to a tuplespace
 - read a tuple matching a template from a tuplespace, leaving it there
 - remove a tuple matching a template from a tuplespace

Tuplespace Overview (Cont'd)

- **Processes generally perform these steps**
 - wait for a tuple matching a given template to appear in a given tuplespace
 - remove the tuple from the tuplespace
 - operate on the tuple
 - create a new tuple describing the result
 - add the new tuple to a tuplespace
 - some other process will operate on that tuple
- **Cooperating processes**
 - processes don't know about or communicate with each other
 - they simply add tuples to and remove tuples from tuplespaces

Uses For Tuplespaces

- **Global persistent communication buffer**
 - tuplespaces persist data as tuples
 - distributed processes can read and write tuples as a way of communicating with each other
- **Lightweight database**
 - tuplespaces are databases and tuples are records
- **Queue manager**
 - simple form of IBM's MQ Series
 - think publish (write tuples) and subscribe (block while waiting for certain tuples to appear)
- **Dynamic computation engine**
 - breaking a complex computation into parts that are each computed when their inputs are available
- **Simulation**
 - model real world processes that have dependencies between each other

Rinda

- Ruby tuplespace implementation
 - based on “Linda”
 - similar to Java’s “JavaSpaces”
 - built on top of dRuby
- To use

```
require 'rinda/tuplespace'
```
- Classes to use
 - tuples are just arrays
 - but are represented as **Rinda::TupleEntry** objects inside Rinda
 - tuplespaces are **Rinda::TupleSpace** objects
 - proxies for communicating with tuplespaces are **Rinda::TupleSpaceProxy** objects

Rinda::TupleSpaceProxy Class

- **Methods**

- **new**

- creates a new `TupleSpaceProxy` for a `TupleSpace` at a given URI

- **write**

- writes a tuple into a `TupleSpace`
 - optional last parameter specifies expiration time in seconds

- **take**

- blocks until a matching tuple appears in a `TupleSpace`, then removes it and returns it
 - optional last parameter specifies timeout in seconds
 - throws `Rinda::RequestExpiredError` if not found within timeout

- **read**

- same as **take**, but doesn't remove tuple from `TupleSpace`

- **read_all**

- reads all matching tuples from a `TupleSpace` without blocking and returns them in an array
 - returns an empty array if no matches are found

in **take** and **read**,
if multiple matches are found,
any one can be returned

Rinda::TupleSpaceProxy Class (Cont'd)

- **Methods (cont'd)**

- **notify**

- notifies a client that one of the following has occurred
 - **'write'** – a tuple was added to a given `TupleSpace`
 - **'take'** – a tuple was taken from a given `TupleSpace`
 - **'delete'** – a tuple was lost from a given `TupleSpace` because it was overwritten or it expired
 - **'close'** – the notification request expired (timed out)

what does
“overwritten”
mean?

Rinda Example

based on example in Pickaxe 2

- **common.rb**

- shared by whiteboard, client and server

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

```
WHITEBOARD_URI = 'druby://localhost:1919'
```

- **whiteboard.rb**

- run this first

```
require 'common'
```

```
DRb.start_service(WHITEBOARD_URI, Rinda::TupleSpace.new)
```

```
DRb.thread.join # don't exit
```

Rinda Example (Cont'd)

- **server.rb**

- run this second

```
require 'common'

DRb.start_service

ts = Rinda::TupleSpaceProxy.new(
  DRbObject.new_with_uri(WHITEBOARD_URI))

# Note that "-" must be the first character in the character class.
# Otherwise it will be interpreted as a range delimiter.
operation_pattern = %r{^[~+*/]$}

loop do
  op, p1, p2 = ts.take [operation_pattern, Numeric, Numeric]
  ts.write ['result', p1.send(op, p2)]
end
```


Rinda Example (Cont'd)

- **client.rb**

- run this third

```
require 'common'
```

```
DRb.start_service
```

```
ts = Rinda::TupleSpaceProxy.new(
```

```
  DRbObject.new_with_uri(WHITEBOARD_URI))
```

```
ts.write ['+', 19, 3]
```

```
# nil in a tuple template means accept any type
```

```
result = ts.take ['result', Numeric]
```

```
puts result[1] # just want 2nd piece of data in result tuple
```

Issue

the result tuple may actually be intended for a different client; consider tagging tuples with a client id

Rinda Documentation

- Where Ruby Really Sparkles
 - Dave Thomas
 - http://www.linux-mag.com/2002-09/ruby_01.html
- There's really very little documentation on Rinda!