RAFDA Run-Time (RRT) Beginner's Guide v1.0

Scott M. Walker
RAFDA Project
School of Computer Science
University of St Andrews
http://www-systems.dcs.st-and.ac.uk/rafda/

Abstract

The RAFDA Run-Time is a Java based middleware system that minimizes the programmer effort required to create a distributed application. A single line of code can make instances of any class remotely accessible or allow clients to obtain remote references to remotely accessible object transparently.

The sections entitled **Deploying a Web Service** and **Accessing a Deployed Object Remotely** on pages 3 to 5 can be used as a **Quick Start Guide**.

Downloads of all software and user guides along with complete JavaDoc can be found at the website **http://www-systems.dcs.st-and.ac.uk/rafda/**

Contents

Introduction **3**

Installation **3**

Deploying a Web Service **3**
Limitations of the standard class loader 5

Remotely Accessing a Deployed Object **5**

Pass-by-value vs. Pass-by-reference **6**
Automatic Deployment Semantics 8

Smart Proxies **8**

Exceptions **9**

Advanced Server and Client **9**

Custom Class Loader **9**

Deploying a Single Object with Multiple Deployment interfaces **10**

Transmission Policy **11**
Class Policy Example 12
Method Transmission Policy and Smart Proxy Example 15

Configuration **18**
How the RRT chooses which network interface and port to bind to 18
Firewall 18
Socket Factories 19
Socket Listener 19

Introduction

The RAFDA Run-Time (RRT) is a middleware system that permits the arbitrary exposure of application objects to remote access irrespective of their class. It provides a remote reference scheme that allows inter-address-space references between arbitrary application objects in order to preserve non-distributed application semantics. The RRT separates the roles of application developer and application distributor by providing a policy framework that allows control over the parameter passing semantics used during remote method call without modification to application source. The RRT is a standalone middleware system for the development of distributed systems and addresses the requirements of three distinct use case scenarios:

- Developing new distributed applications.
- Introducing distribution into non-distributed applications.
- Developing and deploying Web Services.

This guide describes RRT version 1.0, which is available at (http://www-systems.dcs.st-and.ac.uk/rafda/). It permits the creation of distributed applications and its capabilities are illustrated by the various examples in this guide. This guide does not describe the RAFDA transformation tools, which are distributed separately. See the document *Performing Automatic Application Transformation using the RAFDA Tools* for additional information about using the RRT in conjunction with the transformation tools and how distribution polices are defined.

Installation

The RRT is distributed in the JAR file *rrt.jar* which must be present in the classpath along with the following dependencies:

- The *Java 2 Platform, Standard Edition*. We use v1.4.2 available at (http://java.sun.com/j2se/1.4.2/download.html).
- The *Byte Code Engineering Library (BCEL)*. We use v5.1 available at (http://jakarta.apache.org/site/binindex.cgi).
- The *Java Uuid Generator (JUG)*. We use v1.1.1 available at (http://www.doomdark.org/doomdark/proj/jug/curr/jug.jar).
- The *University of St Andrews Dynamic Java Compiler*. We use v1.4 available at (http://www-ppg.dcs.st-and.ac.uk/Java/DynamicCompilation/javacompiler.jar).
    - The Dynamic Compiler requires *tools.jar*, which is distributed with the J2SE. It can usually be found in the JAVA_HOME/lib folder but is not in the classpath by default.

All methods within the RRT are accessed via the *RafdaRunTime* class (*uk.ac.stand.dcs.rafda.rrt.RafdaRunTime*) and the full JavaDoc API for this class can be found in the *rrt.jar*. In this guide, several examples are used to illustrate how to use the RRT. All examples are included in the *rrt.jar* distribution.

Deploying a Web Service

This section shows how an instance of an arbitrary class, *Person*, can be deployed as a Web Service. The *Person* code is as follows:

```
public class Person {
    public String name = null;

    public Person(String name) {
        this.name = name;
    }

    public Person() {
    }
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

The *Person* class has not been written with any consideration for remote access; however, an instance of a *Person* can be deployed as a Web Service using the RRT. To deploy an object the following is required:

- A reference to the object to deploy
- A list of the methods within this object that should be remotely accessible
- A name for the Web Service

To deploy an object, the server makes a single call to the *deploy* method in the *RafdaRunTime* class:

```
public static void deploy(Class interfaze, Object object, String name) throws Exception
```

The first argument specifies the methods within the object that should be remotely accessible. A list of methods can be unambiguously specified using a Java class or interface in the form of a Java *java.lang.Class* object (Note that interfaces in Java are still represented by *java.lang.Class* objects). It is important to note that the deployed object does *not* need to extend or implement the class or interface specified by the *java.lang.Class* object. This type is known as the *deployment interface* though it can be a class or an interface. If a class is used then the code in the class is ignored; only the method signatures are relevant. Remote reference holders believe the type of the deployed object is that of the deployment interface, irrespective of their actual classes! In this example, all methods in the object are to be deployed and so the object's own class can be used. The second argument is a reference to the object to be deployed and the third is a name for the deployed Web Service. The server code follows:

```
import uk.ac.stand.dcs.rafda.rrt.RafdaRunTime;

public class Server {
    public static void main(String[] args) throws Exception {
        Person p = new Person("Scott");
        RafdaRunTime.deploy(Person.class, p, "somePerson");
    }
}
```

When this server class is compiled and run, the following output is produced by the RRT:

```
CLI UI ** RRT started on anya at port 5001 [Socket Listener on port 5001]
CLI UI ** The current class loader is sun.misc.Launcher$AppClassLoader@53ba3d
CLI UI ** Deployed instance of (example1.Person) with name (somePerson). UUID = 59c78a73-
01e7-4b8c-8045-b3d77c0c8d11
```

The parts in bold will change from machine to machine. This output was produced by an RRT running on a machine called *anya* on port *5001*. Indeed, all example output in this guide is generated from a machine called *anya* using the default port configuration. The manner in which the RRT determines which network interface and port to bind to, and how it can be overridden, is described later. The hexadecimal string is an automatically generated UUID. Note that this output states the class of *Person* to be *example1.Person*, indicating that the *Person* class is in package *example1*. This reflects the actual output that will be obtained when running the examples. The package statements have been omitted from the code here for brevity.
The Web Service is now deployed and ready for use via the following URLs:

```
http://anya:5001/somePerson
```

and:

```
http://anya:5001/59c78a73-01e7-4b8c-8045-b3d77c0c8d11
```

which can be more generally specified as:

```
http://<hostname>:<port>/<name>
```

and:

```
http://<hostname>:<port>/<uuid>
```

The above code is included in example 1 supplied with the *rrt.jar*.

Limitations of the standard class loader

If the RAFDA custom class loader is not employed, as described later, then some limitations apply to the objects that can be deployed:

- A deployed object must have a no-arguments constructor.
- A final class cannot be used as the deployment interface.
- No direct field access should be performed on a remote reference; instead, get and set methods must always be used. Such direct field access operations are possible but the semantics are undefined.

The first limitations can be overcome using the RAFDA custom class loader.

Remotely Accessing a Deployed Object

A client can obtain a remote reference to a deployed object based on its name and the name of the machine and port on which the RRT that deployed it is running. The client makes a call to the *RafdaRunTime* method *getObjectByName()* and specifies the host name and port of the RRT to connect to, along with the name of the deployed object. The returned object can be cast into the correct type. Note that unlike many middleware systems, the client does not need to refer to remote objects using interface types. The class of the remote reference is that of the deployment interface with which the remotely referenced object was deployed, not the class of the deployed object!

```
public static Object getObjectByName(String host, int port, String name) throws Exception
```

Note that the RRT binds to a specific interface on its host, which is displayed at startup:

```
CLI UI ** RRT started on hostName at port ...
```

During a call to *getObjectByName()*, it is this hostname that must be specified; generally the RRT does *not* bind to the *localhost* interface. Network interface bindings are discussed later.
The following client code obtains a remote reference to the *Person* object deployed by the server, above and calls some methods on it. To the client it is indistinguishable whether the referenced object is local or remote.

```
import uk.ac.stand.dcs.rafda.rrt.RafdaRunTime;

public class Client {
    public static void main(String[] args) throws Exception {
        Person p = (Person) RafdaRunTime.getObjectByName("anya", 5001,
                "somePerson");
        System.out.println("p1 name is " + p.getName());
        p.setName("Stuart");
        System.out.println("p1 name is " + p.getName());
```

```
    }
}
```

The above code is also included in example 1 supplied with the *rrt.jar*.

Pass-by-value vs. Pass-by-reference

During remote method call, the arguments and return values that cross address-space boundaries may be primitive types or reference types. Primitive types are immutable in Java and so are always passed by-value. Interface types can be passed across the network by-value or by-reference though Java local semantics are pass-by-reference. The RRT is capable of passing objects both by reference and by value.

Conventional Web Service semantics are pass-by-value and so if the client accessing the a deployed object in an RRT is a client implemented using another Web Service technology the RRT will employ standard pass-by-value Web Service semantics. The RRT can serialize instances of any class for transmission across the network using automatically generated custom serializers. If an object is to be transmitted by-value then some limitations apply:

- The object cannot make use of native code.
- All fields must be publicly accessible. This limitation can be overcome using the RAFDA custom class loader, discussed later.

If the client is also executing within an RRT, then the system behaves as a Distributed Object Model and will default to pass objects by-reference in order to preserve local semantics. The parameter passing semantics are completely under the control of the programmer and the manner in which they are altered is described later. If an object is passed by-reference then it must be remotely accessible to the remote reference holder. The RRT must therefore automatically deploy objects that are passed by-reference. The semantics of this are discussed after the following example.

This example illustrates the pass-by-reference semantics employed when using the RRT as a Distributed Object Model. The server deploys a *Person* instance as before, though in this case the *Person* holds a reference to a *Dog* instance. The revised *Person* class follows.

```
public class Person {

    public String name = null;

    public Dog dog = null;

    public Person(String name, Dog dog) {
        this.name = name;
        this.dog = dog;
    }

    public Person() {
    }

    public String getName() {
        return name;
    }

    public void setName(String string) {
        name = string;
    }

    public Dog getDog() {
        return dog;
    }

    public void setDog(Dog d) {
        dog = d;
    }

}
```

The *Dog* class is as follows:

```
public class Dog {
    public String name = null;
    public int age = 0;

    public Dog() {
    }

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getDogName() {
        return name;
    }

    public void setDogName(String n) {
        name = n;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int a) {
        age = a;
    }

    public void sayHello() {
        System.out.println("Dog is " + name + " aged " + age);
    }
}
```

An instance of a *Person* with a *Dog* is instantiated in a server as follows:

```
import uk.ac.stand.dcs.rafda.rrt.RafdaRunTime;

public class Server {
    public static void main(String[] args) throws Exception {
        Person p = new Person("Scott", new Dog("serverDog", 10));
        RafdaRunTime.deploy(Person.class, p, "somePerson");
    }
}
```

The following client is created to access the remote *Person*, obtain a reference to its *Dog* instance and perform some operations on that *Dog*.

```
import uk.ac.stand.dcs.rafda.rrt.RafdaRunTime;

public class Client {

    // Edit this line
    private static final String HOSTNAME = "anya";

    public static void main(String[] args) throws Exception {
        Person p = (Person) RafdaRunTime.getObjectByName(HOSTNAME, 5001,
                "somePerson");
        p.getDog().sayHello();
        Dog d = new Dog("clientDog", 5);
        p.setDog(d);
        p.getDog().sayHello();
        System.out.println("dog is " + d);
        System.out.println("dog is " + p.getDog());
    }
}
```

This client first accesses the *Person's* existing *Dog*, which is server-side. It then creates a new *Dog* and sets the Person's *Dog* to be this new *Dog*. The server output follows and has been annotated to indicate what is occurring:

```
CLI UI ** RRT started on anya at port 5001 [Socket Listener on port 5001]
CLI UI ** The current class loader is sun.misc.Launcher$AppClassLoader@53ba3d
CLI UI ** Deployed instance of (example2.Person) with name (somePerson). UUID = 5121ac11-
6826-4a50-bfe2-4f64bc5a7ddd
```

The Person instance has been deployed in the server.

```
CLI UI ** Deployed instance of (example2.Dog) with no name. UUID = 79ded07e-3c59-464d-
bd77-9970f0751dcc
```

The client has called getDog() and so the Dog is automatically deployed and passed by reference.

```
Dog is serverDog aged 10
```

The sayHello() method is called on the deployed Dog.

The annotated client output follows:

```
CLI UI ** Port 5001 is busy. Automatically choosing port: Trying 5002
CLI UI ** RRT started on anya at port 5002 [Socket Listener on port 5002]
CLI UI ** The current class loader is sun.misc.Launcher$AppClassLoader@53ba3d
```

The client has accessed the remote Dog and called sayHello(), neither of which results in output here.

```
CLI UI ** Deployed instance of (example2.Dog) with no name. UUID = fb932334-ee83-4bd4-
914f-de8c4831ca67
```

The client has instantiated a new Dog and has called setDog(). This new Dog instance must then be automatically deployed here if it is to be passed by-reference.

```
Dog is clientDog aged 5
```

The getDog method is called on the remote Person and sayHello() is called on the Dog, which is in the local address space.

```
dog is example2.Dog@c4fe76
dog is example2.Dog@c4fe76
```

It can be seen that the local reference to the new Dog and the Person's dog field reference exactly the same Java object.

This code is example 2 supplied with the *rrt.jar*.

Automatic Deployment Semantics

When automatically deploying an argument or return value during a remote method call, the RRT decides which deployment interface is appropriate using the following rules.

1. If the object is already deployed using its own class as a deployment interface, then no steps need to be taken.
2. The object may already be deployed using another deployment interface or not deployed at all. The RRT checks the signature of the argument or return value in the method being called and deploys the object using that signature class as the deployment interface. For example, if a method *putFish(Fish f)* is called and an instance of *BigFish* (where *BigFish* is a subtype of *Fish*) is supplied then the *BigFish* instance will be automatically deployed using *Fish* as the deployment interface.

Smart Proxies

Smart proxies allow the remote references to cache some of the deployed object's state locally, thus avoiding the cost of a remote method call. The fields to be cached in the smart proxy are defined server-side using this method in the *TransmissionPolicyManager* class (*uk.ac.stand.dcs.rafda.rrt.policy.transmission.TransmissionPolicyManager*).

```
public static void setFieldToBeCached(String className, String field);
```

For example, the following indicates that the field named *dog* in the *Person* class should be cached:

```
TransmissionPolicyManager.setFieldToBeCached("Person", "dog");
```

ensures that the field *dog* in class *Person* will be cached in the remote reference. It is essential that the remote reference holder access the field using the get and set accessor methods that adhere exactly to the following naming scheme:

```
public <fieldClass> get<fieldNameWithCaptializedFirstLetter>();
public void set<fieldNameWithCaptializedFirstLetter>(<fieldClass>);
```

For example:

```
public DogImpl getDog();
public void setDog(DogImpl);
```

No automatic coherency control is performed by the RRT and full responsibility for ensuring correct application semantics remains with the developer. Changes to the caching policy do not affect existing references, which continue to cache the same set of fields. Clients can obtain an updated reference that caches different fields, to reflect this new policy, directly from the RRT using *getObjectByName()* in the normal manner.

Exceptions

The RRT will propagate exceptions across the network. The programmer can specify whether or not exceptions should be handled by the RRT or propagated back into the application as follows. If the deployment interface specifies that a method throws a *java.lang.Exception* then all exceptions, including network related expectations, are propagated back into the application by the RRT. In this case, the deployment interface and the actual class of the deployed object may differ as the former is permitted to specify that a method throws a *java.lang.Exception* when the actual method does not. As a result, unchecked Exceptions can be propagated across the network. If the deployment interface does not indicate tjat a method throws a *java.lang.Exception*, then any Exceptions that occur during its execution are consumed by the RRT.

Advanced Server and Client

Example 3 illustrates a more advanced server and client that make use of remote reference, smart proxies, transmission policy (discussed later) and the propagation of exceptions across the network.

Custom Class Loader

The RAFDA custom class loader is found in class *RafdaClassLoader* (*uk.ac.stand.dcs.rafda.rrt.infrastructure.RafdaClassLoader*) and will transform application classes at load time to ensure that all fields are public, to enable the serializer to work, and that no classes are final, to allow the deployment of an object using a final class as deployment interface.
To use the custom class loader, all RRTs that cooperate in running a single distributed application must be running the RAFDA class loader. If some RRTs use the RAFDA class loader and some do not then semantics are unpredictable. The following Java Virtual Machine argument must be used when running clients and servers (and all other JVMs that participate in a single application) to make use of the RAFDA custom class loader:

```
-Djava.system.class.loader=uk.ac.stand.dcs.rafda.rrt.infrastructure.RafdaClassLoader
```

Example 4 supplied with the *rrt.jar* illustrates the use of the custom RAFDA class loader.

Deploying a Single Object with Multiple Deployment interfaces

A single object can be deployed multiple times under different deployment interfaces. Provided there is a structurally equivalent method in the class of the object to be deployed for every method in the deployment interface, the deployment will succeed. The deployed object does not need to be of a class that extends or implements the deployment interface. Example 5 consists of a *Student* deployed in multiple different fashions. The *Student* class appears as follows:

```
public class Student implements StudentInterface {
    public String getName() {
        return "Bob";
    }

    public int getMatricNumber() {
        return 1234567;
    }

    public boolean hasTail() {
        return false;
    }
}
```

Several interfaces are defined, all of which contain methods present in the *Student* class:

```
public interface StudentInterface {
    public String getName();
    public int getMatricNumber();
}

public interface PersonInterface {
    public String getName();
}

public interface MammalInterface {
    public String getName();
    public boolean hasTail();
}
```

A *Student* is instantiated and deployed three times using different deployment interfaces. A single object appears as if it is three separate Web Services as it is deployed multiple times using these different deployment interfaces (actually, interfaces in this case). Note that the *Student* does *not* need to implement the interface under which it is deployed.

```
import uk.ac.stand.dcs.rafda.rrt.RafdaRunTime;

public class Server {
    public static void main(String[] args) throws Exception {
        Student s = new Student();
        RafdaRunTime.deploy(PersonInterface.class, s, "person");
        RafdaRunTime.deploy(MammalInterface.class, s, "mammal");
        RafdaRunTime.deploy(StudentInterface.class, s, "student");
    }
}
```

The client then obtains three distinct remote references to the three different Web Services. To the client each reference appears to reference a distinct that is typed the same as the respective deployment interface.

```
import uk.ac.stand.dcs.rafda.rrt.RafdaRunTime;

public class Client {
    public static void main(String[] args) throws Exception {
        PersonInterface p = (PersonInterface) RafdaRunTime.getObjectByName(
                "anya", 5001, "person");
```

```
        MammalInterface m = (MammalInterface) RafdaRunTime.getObjectByName(
                "anya", 5001, "mammal");
        StudentInterface s = (StudentInterface) RafdaRunTime.getObjectByName(
                "anya", 5001, "student");
        System.out.println(p.getName());
        System.out.println(m.hasTail());
        System.out.println(s.getMatricNumber());
    }
}
```

Transmission Policy

The complete transmission policy framework is too complex to be fully described here. See **A Middleware System that Promotes Reuse by Separating Transmission Policy from Implementation** [[icdcs]] for a more complete explanation. This sections explains the outline of the transmission policy framework and shows some examples.

During remote method calls, objects are passed across address space boundaries as arguments and return values. When the RRT is used as a Distributed Object Model, the programmer can control the parameter passing semantics, known as the transmission policy, in order to determine whether objects cross address-space boundaries by-reference or by-value. The transmission policy dictates the manner in which objects are encoded for transmission and decides which parameter passing semantics will be employed during remote method calls.

In order to define the transmission policy for an application, the programmer specifies a series of policy rules. There are four kinds of policy rule:

- Parameter policy rules
- Method policy rules
- Return Value policy rules
- Class policy rules

Parameter policy rules are associated with individual method parameters and they indicate how particular method arguments should be passed across address-space boundaries during a call to the specified method. They allow fine-grain control over the transmission policy that is applied to the parameters of a method. For example, a parameter policy rule might specify that during a call to a particular method, the second parameter should be passed-by-value.

Method and Return Value policy rules are associated with methods as a whole and they specify how return values from methods should be passed across address-space boundaries. For example, a method policy rule might specify that during a call to a particular method, the return value should be passed-by-reference. Additionally, they allow a single transmission policy to be associated with all parameters of a method, avoiding the need to specify a parameter policy rule for each. For example, a method policy rule might specify that during a call to a particular method, all parameters should be passed-by-value. Policies associated with parameters and methods take a depth argument. This indicates how deep into the closure of an object the policy should be applied, after which the default pass-by-reference semantics are used.

Class policy rules are associated with classes and they indicate how instances of classes should be passed across address-space boundaries. For example, a class policy rule might specify that all instances of a particular class are passed-by-value. Each class policy rule applies to exactly one class and does not apply to sub-classes of that class. Class policy rules are applied based on the actual classes of the arguments, rather than those specified in the method signature, which may be super-classes of the arguments.

Policy rules apply only in the address space in which they are specified, and they apply to all objects in that address space. Policy rules can be specified either statically or dynamically. To specify policy rules statically, a library class programmer can specify the policy rules in the class's initialization code. For example, in Java, policy rules may be specified in the static initializer and are then active from class load time.

In contrast, an application programmer may specify or change policy rules at any point during application execution, in which case they come into force immediately—thus allowing for dynamic adaptation of the application.
Valid transmission policies are defined in the *PolicyType* class and are:

- By-reference
- By-value
- Undefined

Transmission policies are defined by calling into the *TransmissionPolicyManager* class which contains a series of static methods. setClassPolicy()

- setMethodPolicy()
- setParamPolicy()
- setReturnValuePolicy()

These classes are fully described in the JavaDoc supplied with the RRT.
The RRT effectively prioritizes the rules as follows:

1. Parameter policy rule
2. Method policy rule
3. Class policy rule
4. Default policy

However, each rule can also be defined by the programmer as *non- overridable* and so the RRT actual prioritizes the rules as follows:

1. Parameter policy rule (non-overridable)
2. Method policy rule (non-overridable)
3. Class policy rule (non-overridable)
4. Parameter policy rule (overridable)
5. Method policy rule (overridable)
6. Class policy rule (overridable)
7. Default policy

In this ordering, it is possible that a non-overridable class policy rule will still be overridden by a non-overridable method policy.

Class Policy Example

The following shows example 6 in which several Person objects are placed into a House. The House appears as follows:

```
import java.util.HashSet;
import java.util.Iterator;

public class House {
    HashSet persons = new HashSet();

    public void showContents() {
        for (Iterator i = persons.iterator(); i.hasNext();) {
            Person p = ((Person) i.next());
            System.out.print(p.getName());
            if (p.getFriend() != null) {
                System.out.println(" has a friend called "
                        + p.getFriend().getName());
            } else {
                System.out.println(" has no friend");
            }
        }
        System.out.println();
    }

    public void addPerson(Person p) {
        persons.add(p);
    }
}
```

The Person class is as follow:

```
public class Person {
    public String name = null;

    public Person friend = null;

    public Person(String name) {
        this.name = name;
    }

    public Person() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setFriend(Person friend) {
        this.friend = friend;
    }

    public Person getFriend() {
        return friend;
    }
}
```

The House is deployed as normal:

```
import uk.ac.stand.dcs.rafda.rrt.RafdaRunTime;

public class Server {
    public static void main(String[] args) throws Exception {
        RafdaRunTime.deploy(House.class, new House(), "house");
    }
}
```

The client then accesses the house, applying different class policies to the *Person* class:

```
import uk.ac.stand.dcs.rafda.rrt.RafdaRunTime;
import uk.ac.stand.dcs.rafda.rrt.policy.transmission.PolicyType;
import uk.ac.stand.dcs.rafda.rrt.policy.transmission.TransmissionPolicyManager;

public class Client {
    public static void main(String[] args) throws Exception {
        Person p1 = new Person("Scott");
        Person p2 = new Person("Stuart");
        Person p3 = new Person("Alvaro");
        Person p4 = new Person("Kath");

        p1.setFriend(p3);
        p2.setFriend(p4);

        House house = (House) RafdaRunTime.getObjectByName("anya", 5001,
                "house");

        TransmissionPolicyManager.setClassPolicy(Person.class.getName(),
                PolicyType.BY_REFERENCE, true);

        house.addPerson(p1);

        TransmissionPolicyManager.setClassPolicy(Person.class.getName(),
                PolicyType.BY_VALUE, true);
```

```
        house.addPerson(p2);

        // Do some output

        System.out.println("p1 = " + p1.getName());
        System.out.println("p2 = " + p2.getName());
        System.out.println("p3 = " + p3.getName());
        System.out.println("p4 = " + p4.getName());
        System.out.println();
        house.showContents();

        p1.setName("He");
        p2.setName("She");
        p3.setName("Him");
        p4.setName("Her");

        System.out.println("p1 = " + p1.getName());
        System.out.println("p2 = " + p2.getName());
        System.out.println("p3 = " + p3.getName());
        System.out.println("p4 = " + p4.getName());
        System.out.println();
        house.showContents();

        System.exit(0);
    }
}
```

The client-side output look like this:

```
CLI UI ** Port 5001 is busy. Automatically choosing port: Trying 5002
CLI UI ** RRT started on anya at port 5002 [Socket Listener on port 5002]
CLI UI ** The current class loader is sun.misc.Launcher$AppClassLoader@53ba3d
CLI UI ** Deployed instance of (example7.Person) with no name. UUID = 2e497426-6dbf-435d-
8c36-aa9968cffadc
p1 = Scott
p2 = Stuart
p3 = Alvaro
p4 = Kath
```

Initally the Person names are set to as shown...

```
p1 = He
p2 = She
p3 = Him
p4 = Her
```

But then the client performs a series of setName() calls and changes the names.

The server-side output look like this:

```
CLI UI ** RRT started on anya at port 5001 [Socket Listener on port 5001]
CLI UI ** The current class loader is sun.misc.Launcher$AppClassLoader@53ba3d
CLI UI ** Deployed instance of (example7.House) with name (house). UUID = 30baf18a-5e8b-
4352-b958-069480cc8b8c
Scott has a friend called Alvaro
Stuart has a friend called Kath
```

The output of the first call to showContents(). The names are as expected.

```
He has a friend called Him
Stuart has a friend called Kath
```

The output of the second call to showContents(). This illustrates that p1 and p3 were passed by-reference, as the changes made to the names client-side are reflected here, while p2 and p4 must have been passed by-value because they retain the old names.

In this example, Persons p1, p2, p3 and p4 exist client-side. During insertion into the remote house, p1, and hence its referenced friend p3, are passed-by-reference while p2, and hence its referenced

friend p4, are passed-by-value, as dictated by the transmission policy. When changes are made to the names of the Person instances on the client side, only those Person objects that were passed by-reference show the changes. Those that were passed by-value are unchanged.

Method Transmission Policy and Smart Proxy Example

In this example, which is example 7 in the *rrt.jar*, method transmission policies are used. Initially a *Person* class is defined, in which it is specified that the *dob* (date of birth) field should be cached in the remote reference, since the date of birth of a *Person* cannot change. The methods in *Person* have been extended to print a message to standard out when called in order to help illustrate in which address-space the *Person* objects exists.

```
import uk.ac.stand.dcs.rafda.rrt.policy.transmission.TransmissionPolicyManager;

public class Person {

    static {
        TransmissionPolicyManager.setFieldToBeCached("example7.Person", "dob");
    }

    public String name = null;
    public int dob = 0;

    public Person() {
    }

    public Person(String name, int dob) {
        this.name = name;
        this.dob = dob;
    }

    public int getDob() {
        System.out.println("Call to getDob() of (" + name + ", " + dob + ", "
                + hashCode() + ")");
        return dob;
    }

    public String getName() {
        System.out.println("Call to getName() of (" + name + ", " + dob + ", "
                + hashCode() + ")");
        return name;
    }

    public void setName(String name) {
        System.out.println("Call to setName() of (" + name + ", " + dob + ", "
                + hashCode() + ")");
        this.name = name;
    }

    public String toString() {
        return "Person is " + name + " born " + dob + " [" + hashCode() + "]";
    }
}
```

A *PersonManager* class, which is capable of performing some operations on *Person*, is defined.

```
import java.util.HashSet;
import java.util.Iterator;

import uk.ac.stand.dcs.rafda.rrt.policy.transmission.PolicyType;
import uk.ac.stand.dcs.rafda.rrt.policy.transmission.TransmissionPolicyManager;

public class PersonManager {
    private HashSet persons = new HashSet();

    static {
        TransmissionPolicyManager.setMethodPolicy("example7.PersonManager",
```

```
                "addPersonToRecords", PolicyType.BY_REFERENCE, -1, true);
        TransmissionPolicyManager.setReturnValuePolicy(
                "example7.PersonManager", "getPersonFromRecords",
                PolicyType.BY_REFERENCE, true);
        TransmissionPolicyManager.setMethodPolicy("example7.PersonManager",
                "getFirstInitial", PolicyType.BY_VALUE, -1, true);
        TransmissionPolicyManager.setMethodPolicy("example7.PersonManager",
                "isFirstPersonOlder", PolicyType.BY_VALUE, -1, true);
        TransmissionPolicyManager.setMethodPolicy("example7.PersonManager",
                "returnOlderPerson", PolicyType.BY_REFERENCE, -1, true);
    }

    public void addPersonToRecords(Person p) {
        persons.add(p);
    }

    public Person getPersonFromRecords(String name) {
        for (Iterator i = persons.iterator(); i.hasNext();) {
            Person p = ((Person) i.next());
            if (p.getName().equals(name))
                return p;
        }
        return null;
    }

    public String getFirstInitial(Person p) {
        return p.getName().substring(0, 1);
    }

    public boolean isFirstPersonOlder(Person p1, Person p2) {
        if (p1.getDob() < p2.getDob())
            return true;
        else
            return false;
    }

    public Person returnOlderPerson(Person p1, Person p2) {
        if (p1.getDob() < p2.getDob())
            return p1;
        else
            return p2;
    }
}
```

Consider the transmission policy defined here. The *addPersonToRecords()* has a policy that states that its arguments should be passed by-reference and *getPersonFromRecords()* has a policy that states that its return value should be passed by-reference. Conversely, the *getFirstInitial()* is defined as taking a *Person* object by-value. Two different but functionally similar methods, *isFirstPersonOlder()* and *returnOlderPerson()*, are specified though they use different policies.
The *PersonManager* is deployed in the usual manner:

```
import uk.ac.stand.dcs.rafda.rrt.RafdaRunTime;

public class Server {
    public static void main(String[] args) throws Exception {
        PersonManager pm = new PersonManager();
        RafdaRunTime.deploy(PersonManager.class, pm, "personManager");
    }
}
```

The client makes several calls to the different *PersonManager* methods and the different transmission policies are automatically enforced.

```
import uk.ac.stand.dcs.rafda.rrt.RafdaRunTime;

public class Client {
    public static void main(String[] args) throws Exception {
```

```
        RafdaRunTime.startConnectionListener();

        PersonManager pm = (PersonManager) RafdaRunTime.getObjectByName(
                "anya", 5001, "personManager");
        Person p1 = new Person("Scott", 19700101);
        Person p2 = new Person("Alvaro", 19720514);
        System.out.println(p1);
        System.out.println(p2);
        pm.addPersonToRecords(p1);
        System.out.println("getPersonFromRecords: "
                + pm.getPersonFromRecords("Scott"));
        System.out.println("getFirstInitial: " + pm.getFirstInitial(p1));
        System.out.println("isFirstPersonOlder: "
                + pm.isFirstPersonOlder(p1, p2));
        System.out.println("isFirstPersonOlder: "
                + pm.returnOlderPerson(p1, p2));
    }
}
```

The resultant output client-side follows and has been annotated in bold italics to explain what is occurring.

```
CLI UI ** Port 5001 is busy. Automatically choosing port: Trying 5002
CLI UI ** RRT started on anya at port 5002 [Socket Listener on port 5002]
CLI UI ** The current class loader is sun.misc.Launcher$AppClassLoader@53ba3d
```

The RRT has now started

```
Person is Scott born 19700101 [8442367]
Person is Alvaro born 19720514 [22725577]
```

The first two println statements show the object hash codes in squares brackets

```
CLI UI ** Deployed instance of (example7.Person) with no name. UUID = f8312f00-65c9-4ed3-
8b29-8688c3c529a1
```

Person p1 is automatically deployed during the call to addPersonToRecords()

```
Call to getName() of (Scott, 19700101, 8442367)
getPersonFromRecords: Person is Scott born 19700101 [8442367]
```

During a call to getPersonFromRecords the Person is passed by reference and so the remote PersonManager's call to getName() can be seen. The resultant reference is to p1, as seen by the hash code.

```
getFirstInitial: S
In contrast, the call to getFirstInitial() takes the Person by value and so no call is
seen to the local getName() method. Instead, the getName() call can be seen below in the
server output.
isFirstPersonOlder: true
```

Again, both Person objects are passed by value and so no calls to getDob() are seen here. Instead these calls appear in the server output below.

```
CLI UI ** Deployed instance of (example7.Person) with no name. UUID = 332b29f0-0257-434a-
93ce-8023b76e24b4
isFirstPersonOlder: Person is Scott born 19700101 [8442367]
```

Finally, p2 must be deployed as both Person objects are passed by reference to the isFirstPersonOlder() method. The returned reference is to p1, as seen by the hash code.

The server output follows and is again annotated:

```
CLI UI ** RRT started on anya at port 5001 [Socket Listener on port 5001]
CLI UI ** The current class loader is sun.misc.Launcher$AppClassLoader@53ba3d
CLI UI ** Deployed instance of (example7.PersonManager) with name (personManager). UUID =
47131ddc-3848-4486-b862-9ce244b59b6b
```

The RRT has started and deployed the PersonManager.

```
Call to getName() of (Scott, 19700101, 30844270)
```

This call to getName() is performed on a local copy of p1 during the call to getFirstInitial().

```
Call to getDob() of (Scott, 19700101, 28336193)
Call to getDob() of (Alvaro, 19720514, 8344960)
```

Here calls to getDob() on two local copies of p1 and p2 during the call to isFirstPersonOlder() can be seen.

Configuration

Several aspects of RRT configuration can be overridden by the programmer. Configuration options must be overridden before the RRT starts running. The RRT starts when an object is deployed, either directly by the programmer or automatically as the result of a remote method call. It is suggested that configuration be performed in a static initializer in the class containing the application's *main()* method.

How the RRT chooses which network interface and port to bind to

By default, the RRT binds to the network interface returned by the following Java library class call, the semantics of which are defined in the Java API documentation.

```
InetAddress.getLocalHost();
```

Typically this method will not bind to *localhost*, instead using a DNS host name. The programmer can override the network interface to which the RRT binds at any time before the RRT starts by calling the setHost() method.

```
public static void setHost(InetAddress host);
```

The RRT uses an automated mechanism to determine which port to bind to, depending on whether the programmer has specified a port explicitly. If the programmer has *not* explicitly stated which port to use then the RRT will use the default port of 5001 if it is available. If 5001 is not available, the RRT will try subsequent ports in increments of one until a free port is found and used.
The programmer explicitly chooses a port using the setPort() method:

```
public static void setPort(int newPort);
```

If a port has been explicitly stated, the RRT will only attempt to bind to this particular port at start time; it will *not* automatically try other subsequent ports. If it cannot bind to the port, it will retry five times at 10-second intervals before failing.

Firewall

A basic firewall exists that allows the programmer to state complete IP addresses or prefixes of IP addressed from which a client may connect. Wild cards or net-masks are not permitted. Valid examples include:

- 138.252.1.1 - this address only
- 138.251 - any address beginning 138.251
- 138.252.0. - any address beginning exactly 138.251.0 i.e. NOT 138.251.1.x
- 127.0.0.1 - localhost

This is done via the *useFirewall()* method:

```
public static void useFireWall(String validAddresses);
```

The string of valid address must be specified in a semi-colon separated list. To specify a range include only the part of the address that is common to all addresses in the range, for example:

```
RafdaRunTime.useFireWall("138.251;127.0.0.1;138.252.0.;138.252.1.1");
```

Socket Factories

The RRT will use conventional Java sockets for inter-address-space communication by default. However, the programmer can associate a particular Socket class with a particular host and port, perhaps to provide encryption, profiling or simulate changing network conditions,.
The programmer must obtain the singleton associated with class RafdaSocketFactory, as follows:

```
RafdaSocketFactory.getSingleton()
```

Subsequently the programmer calls the following method on that singleton:

```
public void associateIpAddressWithSocketClass(String host, int port, String socketClass)
throws Exception;
```

For example, to state that a Socket of class *java.example.EncryptedSocket* should be used when contacting host *anya* on port *5001*, the following is specified:

```
RafdaSocketFactory.getSingleton().associateIpAddressWithSocketClass("anya", 5001, "java.
example.EncryptedSocket");
```

Socket Listener

In general, a socket listener allows a user to examine the bytes transmitted on a socket. It reads bytes from the transmitter, displays them and then forwards them to the receiver. Typically, a socket listener listens on port X and propagates all traffic straight through to port Y on some host Z. Clients are instructed to connect to the server via port X on the machine the socket listener runs on, despite the fact that the server runs on port Y on machine Z.
The RRT permits the connection of a socket listener to view all network traffic into and out of an RRT. The observed RRT must be aware of the socket listener, as it must actually bind to port Y but ensure that all remote references state that it actually binds to port X to ensure clients contact it via the socket listener. Port X is known as the *effective port* while port Y is known as the *actual port*.
The programmer must call the following method in the *RafdaRunTime* class:

```
public static void switchSocketListenerOn();
```

The effective port is assumed to be the actual port, as specified using setPort() or determined automatically by the system, plus 50. For example, if the actual port is 5001 then the effective port is 5051.