RWebServices: exposing R to the web

Nianhua Li, Martin Morgan, Seth Falcon, Robert Gentleman
Fred Hutchinson Cancer Research Center
Seattle, WA, USA

7 August, 2007

What is a web service?

- Machine-to-machine interactions
- Client and server, communicating via XML-based SOAP

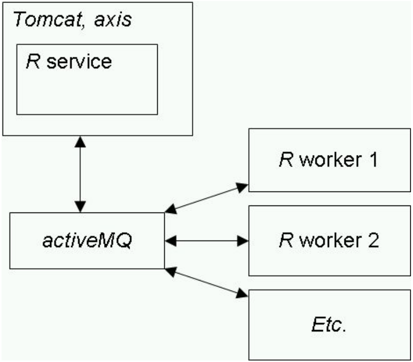
Features we'd like

- Specific methods (not 'all of R')
- Easy to get from R to web service
- Able to handle multiple users

Our web services infrastructure

Flexible, scalable architecture

- One or more workers
- Persistent, so limited service invocation costs
- Server, queue, workers conceptually distinct
- Leveraging Java



Five steps to creating R web services

1. Create and install an R package, using S4 classes for complicated data types and `typeInfo` to specify argument and return types.
2. Create a project with `unpackAntScript`.
3. Map R to Java with `ant map-package`.
4. Create and deploy web services with `ant ws deploy-serv`
5. Start workers, and service requests!

Prerequisites

- R version 2.6.0 or greater.
- RWebServices (depends on SJava, TypeInfo)
- Java Software Development Kit (SDK, *not* the Java Runtime environment, JRE) version `http://java.sun.com` (version 1.5.0 or greater).
- Apache ant `http://ant.apache.org` (e.g., version 1.70).
- Apache ActiveMQ `http://activemq.apache.org` (**version 4.0.2**).
- Apache axis `http://ws.apache.org/axis/`
- Apache tomcat `http://tomcat.apache.org/`

Preparing R functions for web services

Prerequisites

- R compiled with shared-object library support (e.g., on linux `./configure --enable-R-shlib`)
- Java SDK installed, `JAVA_HOME` set. Use `R CMD javareconf` to configure R if Java installed after R.
- R packages TypeInfo, RWebServices, SJava.

Overview

- Identify functionality to be exposed.
- Create data types, if necessary.
- Apply type information to selected functions.
- (Recommended) Create a package to contain functions to be exposed as web services.

Functionality to expose

Why not ‘all of R’?

- Very big security risk.
- Very challenging, need:
  - *Strongly typed* data.
  - Fully parameterized methods.
  - Strongly typed return values.
- Providing a generic statistical calculator is probably not your objective

Instead. . .

- Domain-specific functionality.
- Known or specialized resource requirements.
- (Programmatic) interaction with other software, e.g., in an analytic work flow.

Making R strongly typed: S4

- Provide well-defined data types.
- Information about class structure can be determined programmatically. E.g.

```
> setClass("AClass", contains="numeric")
[1] "AClass"
> setClass("BClass", contains="numeric",
+           representation=representation(
+             transform="character"))
[1] "BClass"
> slotNames("BClass")
[1] ".Data"     "transform"
```

Making R strongly typed: TypeInfo I

- TypeInfo provides a way to define function argument and return values. E.g.,

```
> library(TypeInfo)
> STS <- SimultaneousTypeSpecification
> TS <- TypedSignature
> transform <- function(value, func) {
+     cat("OurWebServices::transform\n")
+     result <- do.call(func, list(value@.Data))
+     b <- new("BClass", result, transform=func)
+     return (b)
+ }
> typeInfo(transform) <-
+   STS(TS(value="AClass",
+          func="character"),
+       returnType="BClass")
```

Making R strongly typed: TypeInfo II

```
> typeInfo(transform)

[SimultaneousTypeSpecification]
  [TypedSignature]
    value: is(value, c('AClass'))  [InheritsTypeTest]
    func: is(func, c('character'))  [InheritsTypeTest]
  returnType: is(returnType, c('BClass'))  [InheritsType
```

(Recommended) Create an R package

Several pieces of information need to be coordinated:

- Functions and methods to be exposed.
- R data types and function signatures.
- Documentation!

We'd also like, perhaps, to

- Expose different collections of functions for different purposes.
- Have some control over how data types map between R and Java

Create an R package I

Key package files

```
[1] "DESCRIPTION"
[2] "man/OurWebServices-package.Rd"
[3] "man/transform.Rd"
[4] "NAMESPACE"
[5] "R/clap.R"
[6] "R/DataClasses.R"
[7] "R/getKidney.R"
[8] "R/transform.R"
```

Create an R package II

The DESCRIPTION file

```
 [1] Package: OurWebServices
 [2] Type: Package
 [3] Title: Example web services
 [4] Version: 1.0
 [5] Date: 2007-08-02
 [6] Author: Martin Morgan
 [7] Maintainer: Martin Morgan <mtmorgan@fhcrc.org>
 [8] Description: These simple R functions are exposed as
 [9]     web services using the RWebServices package
[10] Depends: R (>= 2.6.0), RWebServices, vsn
[11] License: Artistic
```

Create an R package III

The `NAMESPACE` file

```
[1] export(transform, clap, getKidney)
[2] exportClasses(AClass, BClass)
```

Create an R package IV

The R/transform.R file

```
transform <- function(value, func) {
    cat("OurWebServices::transform\\n")
    result <- do.call(func, list(value@.Data))
    b <- new("BClass", result, transform=func)
    return (b)
}
typeInfo(transform) <-
  SimultaneousTypeSpecification(
      TypedSignature(
         value="AClass",
         func="character"),
      returnType="BClass")
```

Map between R and Java

Prerequisites

- As above, and. . .
- ant, ActiveMQ installed; `ANT_HOME`, `JMS_HOME` environment variables set.

Overview

- Create Java classes corresponding to data objects.
- Create a ‘service’ class containing Java methods that invoke corresponding R functions.
- Create test class templates to facilitate testing.
- Create ant scripts to facilitate building customized web services.
- Edit and evaluate ‘local’ test functionality.

Create a project template

- ant scripts for project development (creating R/Java maps; creating and deploying web services, running tests)
- Properties files identify key parameters influencing mapping and service evaluation.

```
% R CMD INSTALL --clean OurWebServices
% echo "library('RWebServices');
     unpackAntScript('OurWebServices_proj')" | R --vanilla
% ls
OurWebServices OurWebServices_proj
% cd OurWebServices_proj
% ls
build.xml  RWebServicesEnv.properties
RWebServicesTuning.properties
```

A basic test

- Does RWebServices know how to talk to Java? Built-in tests move underlying data types back and forth.

```
% cd OurWebServices_proj
% ant rservices-test
...
    [junit] Loading required package: TypeInfo
    [junit] Loading required package: tools
    [junit] Loading required package: SJava
    [junit] Load the Java VM with .JavaInit()
    [junit] Loading required package: TypeInfo
    [junit] Loading required package: tools
    [echo] ===== See the directory './test/output' for more

BUILD SUCCESSFUL
Total time: 9 seconds
```

Map between R and Java I

- Real magic, part 1: create Java representations of R objects and functions (in `src`) and test templates (in `test`)

```
% ant map-package -Dpkg=OurWebServices
...
% ls
... src test
```

- **Warning**: `map-package` over-writes existing `.java` files, e.g., the test cases you have implemented!

Map between R and Java II

Java files in `src/org/bioconductor/`

- `packages.*` represent R data and functions as Java classes.
- `rserviceJms.*` represent the 'front-end' (service) interface, and 'back-end' (worker) implementation.

```
[1] packages/ourWebServices/AClass.java
[2] packages/ourWebServices/BClass.java
[3] packages/ourWebServices/OurWebServicesFunction.java
[4] rserviceJms/services/OurWebServices/OurWebServices.java
[5] rserviceJms/services/OurWebServices/OurWebServicesPrope
[6] rserviceJms/worker/RWorker.java
[7] rserviceJms/worker/RWorkerProperties.java
[8] rserviceJms/worker/RWorkerREnv.java
```

Map between R and Java III

```
 [1] package org.bioconductor.packages.ourWebServices;
 [2]
 [3]     /**
 [4]     * This file was auto-generated by R function
 [5]     * createJavaBean Tue Aug  7 16:12:38 2007.
 [6]     * It represents the S4 Class BClass in R package (
 [7]     */
 [8]
 [9]
[10] public class BClass implements java.io.Serializable  {
[11]     private double[] rData;
[12]     private String[] transform;
[13]
[14]     public BClass() {
[15]         this.rData = new double[]{};
[16]         this.transform = new String[]{};
[17]     }
```

Map between R and Java IV

```
package org.bioconductor.packages.ourWebServices;
import javax.jms.*;
import java.util.*;

public class OurWebServicesFunction {

    /**
    * Java wrapper for R function transform.
    *    ~~ A concise (1-5 lines) description of
    *    what the function does. ~~
    *
    * @param value    ~~Describe value here~~
    * @param func    ~~Describe func here~~
    * @return    ~Describe the value returned If it is
    *    LIST, use \\item{comp1 }{Description of 'comp
    *    \\item{comp2 }{Description of 'comp2'} ...
```

(Recommended) Create unit tests

- E.g., in `test/src/`, find `OurWebServicesTest.java` and modify the method to test the transform method:

```
public void TestTransform() throws RemoteException {
    AClass transform_value =
        new AClass(new double[] {1., 2., 3.});
    String[] transform_func = new String[] {"log"};

    double[] expected =
        new double[] {0.0, 0.6931471805599453, 1.0986122886
    BClass transform_ans =
        new BClass(expected, new String[] {"log"});

    assertEquals(transform_ans,
        binding.transform(transform_value, transform_func))
}
```

Configuring ActiveMQ

- Create an environment variable `JMS_HOME` pointing to ActiveMQ home.
- Edit `$JMS_HOME/conf/activemq.xml` so that the `<broker useJmx="true">` is replaced with `<broker useJmx="true" persistence="false">`.
- Add ActiveMQ configuration information to `RWebServicesEnv.properties`

  jms.host Address of computer running ActiveMQ, e.g., `localhost`

  jms.port Address of JSM service. Default is 61616

Compiling classes and starting ActiveMQ

- Compile the created `.java` files to `.class`.

  ```
  % ant precompile
  ...
  ```

- Open a new console, and start ActiveMQ

  ```
  B% $JMS_HOME/bin/activemq
  ...
  ```

- Open a third console, and start a ‘worker’

  ```
  C% ant start-worker
  ...
  ```

- Leave these consoles open, and return to the main console.

Locally testing the service

- Run the test suite; remember, only *implemented* tests are evaluated!
  - `RWorkerDataTest.java` tests whether, e.g., `AClass.java` can be sent to R, and `BClass.java` can be returned from R.
  - `OurWebServices.java` tests service invocation.

```
% ant local-test
...
BUILD SUCCESSFUL
Total time: 20 seconds
% ls -l test/output
```

- Worker should report `OurWebServices::transform`.
- Additional output in `test/test_output`.

From Java to web service

Prerequisites: Apache axis, deployed into Apache tomcat.

- `CATALINA_HOME` points to tomcat installation directory.
- `axis/webapps/axis` copied to `$CATALINA_HOME/webapps/`
- Start tomcat (`$CATALINA_HOME/bin/startup.sh`) and check required axis components in `http://localhost:8080/axis/happyaxis.jsp`.
- Trouble-shoot by consulting tomcat or axis documentation.

Overview

- Create and install web service infrastructure from Java classes.
- Establish server to receive incoming requests, and ‘workers’ to perform calculations.
- Test.

Creating and installing web services

- The second real magic, with the aid of Apache java2WDLS and WSDL2java

  ```
  % ant ws
  ```

  (same as `ant gen-wsdl mkserver mkclient`).
- Creates directories:

  wsdl Web service description language from Java classes.

  server Web service implementation, connecting to the JMS service.

  client Client interface and test classes.

(Recommended) Create unit test clients for web service

- E.g., in `client/OurWebServices/src/`, find `OurWebServicesServiceTestCase.java` and modify

```
public void test1OurWebServicesTransformw() throws Exceptio
    [...]
    AClass transform_value =
        new AClass(new double[] {1., 2., 3.});
    String[] transform_func = new String[] {"log"};

    double[] expected =
        new double[] {0.0, 0.693147180559945З, 1.0986122886
    BClass transform_ans =
        new BClass(expected, new String[] {"log"});

    assertEquals(transform_ans,
        binding.transform(transform_value, transform_func))
}
```

Testing the service

- Deploy the service to tomcat

  ```
  % $CATALINA_HOME/bin/startup.sh
  % ant deploy-serv
  % $CATALINA_HOME/bin/shutdown.sh
  ```

- Start up ActiveMQ and workers, if necessary.
- Start tomcat.
- Run test

  ```
  % ant web-test
  ```

Five steps to creating R web services

1. Create and install an R package, using S4 classes for complicated data types and `typeInfo` to specify argument and return types.
2. Create a project with `unpackAntScript`.
3. Map R to Java with `ant map-package`.
4. Create and deploy web services with `ant ws deploy-serv`
5. Start workers, and service requests!