Getting Started With River

This document is going to explain how you can use the in-built "simple" services that come with River. More specifically:

- a Lookup Service, the "reggie" implementation
- a JavaSpace, the "outrigger" implementation
- a Transaction Service, the "mahalo" implementation

The instructions assume that you're building from source as checked out from the SVN trunk. Currently this is necessary because the code snippets below use methods and classes which, at time of writing, haven't made it into the latest binary release yet. Having said that, the code you will need in the binary release isn't to far removed from what you'll see below, so you can progress with the binary release if you want to and are happy odifying the code.

If you are going to use the latest release then you will still need to download (from the SVN trunk) some of the Windows BAT files mentioned below. *nix sh scripts of those same batch files will appear shortly.

Environment Setup

The following instructions all assume that you have checked out River to some directory which is throughout referred to as `$RIVER_HOME`. Further, you have changed directory into `$RIVER_HOME/examples/hello`.

Run all scripts from the `hello` directory

- Change directory to $RIVER_HOME
- Use Ant to build River, i.e. `ant all.build`

Running Code Which Uses River Services

Since River uses dynamic code downloading, we need to grant various permissions to our JVM to allow that. In the interest of simplicity, we are going to grant everything every permission. In the real world, this would obviously not be recommended.

To grant these permissions we need to create a policy file:

```
grant {
    permission java.security.AllPermission;
};
```

Create the above file and save it somewhere that it can be easily referenced as a command line argument to some Java you're going to write - such as the working directory that you will run the code from.

Further, your programs which will use River services will need the following JVM arguments.

```
-Djava.security.policy=path/to/policy.all
-Djava.rmi.server.RMIClassLoaderSpi=net.jini.loader.pref.PreferredClassProvider
```

Also, before doing anything with any River code you need to set an appropriate security manager. This needs to be done only once per program.

```
System.setSecurityManager(new RMISecurityManager());
```

Starting the HTTP server

What? Wait, why do I need to do that?

When services get marshalled for sending over the wire, the first part of their stream is the codebase URL. This URL tells the downloading JVM where to load the supporting JARs from. Often (although not always) in the River/Jini world this code base harks back to some HTTP server. There is a simple HTPP server packaged inside River which can be used for this. That's what we're going to start.

See 3.2 How codebase is used in Java RMI for more details.

- Change to directory to `$RIVER_HOME/examples/hello/`
- Execute the script; `scripts/httpd.bat`

Testing it

To test that the HTTP server is running correctly use wget or your web browser on the following URL

- http://localhost:8080/reggie-dl.jar

Starting a Service Registrar (Lookup Service)

What? Wait, why do I need to do that?

Typically, in an environment which uses River/Jini services, a Lookup Service will be used. This Lookup Service is then used by your code to find River/Jini services. Lookup Services always know the current state of the subnet or federated space.

Broadly speaking (although not 100% accurately) you need a Lookup Service in order to be able to find other services to use.

- Change to directory to `$RIVER_HOME/examples/hello/`
- Execute the script; `scripts/jrmp-reggie.bat`

Testing it

There are two ways to find our Lookup Service. Remember that the interface which fulfills the role of a Lookup Service is `ServiceRegistrar`.

Unicast

If we know where our Lookup Service is running, we can use unicast to connect straight to it.

```
LookupLocator ll = new LookupLocator("jini://localhost:4160");
StreamServiceRegistrar sr = ll.getStreamRegistrar();
System.out.println("Service Registrar: " + sr.getServiceID());
```

Assuming you don't get a `NullPointerException` and you do get a service ID written out, then your Lookup Service is running fine.

The `LookupLocator` takes a String representing a "jini URL". This jini URL is made up from the Strings "`jini://`" a hostname and (optionally) "`:`" and a port number. 4160 is the default port and is specified in the Lookup Service configuration file. The scripts in the {{examples/hello}} directory will use the default port unless you have changed it. See DJ.5.5 Address and Port Mappings for TCP and Multicast UDP for more details.

Multicast

If we know only that "some lookup services are on the subnet somewhere" then we can use multicast to find them.

```
DiscoveryListenerManagement dlm = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);

LeaseRenewalManager lrm = new LeaseRenewalManager();
ServiceDiscoveryManager sdm = new ServiceDiscoveryManager(dlm, lrm);

Thread.sleep(500); //need to wait a little bit for the Lookup Service to generate the events to the sdm

ServiceTemplate srTemplate = new ServiceTemplate(null, new Class[] { ServiceRegistrar.class }, null);

ServiceItem[] sis = sdm.lookup(srTemplate, 10, null);
for (ServiceItem si : sis) {
    System.out.println("Service Registrar: "+si.serviceID);
}

dlm.terminate();
```

This approach is slightly different. It is relying the `ServiceDiscoveryManager` to listen to the UDP multicast packets announcing the presence of a Lookup Service. The same rules apply as for unicast on deciding whether or not your Lookup Service is running.

This approach can be modified if you know which hosts on the subnet *might* be running lookup services. You can define and pass this list to the `LookupDiscovery` constructor.

Starting a Java Space

What? Wait, why do I need to do that?

Finding a Lookup Service is only useful if there are some other services on the network which you want to use, so now we're going to start a Java Space, and find that. Using the Java Space is outside the scope of this document, although the "how" is relatively straightforward.

- Change to directory to `$RIVER_HOME/examples/hello/`
- Execute the script; `scripts/jrmp-outrigger-group`

Testing it

To find any kind of service we must first create a template. This is an instance of `ServiceTemplate` which describes the kind of thing we're looking for. As the Jini specification says, nulls and empty arrays as wildcard characters for "any value". So the following code is a template which will match any service which implements the `JavaSpace` interface regardless of the rest of it's characteristics.

```
ServiceTemplate template = new ServiceTemplate(null, new Class[] { JavaSpace.class }, new Entry[0]());
```

Using the unicast and a `ServiceRegistrar` we find the Java Space like this;

```
ServiceMatches sms = sr.lookup(template, 10);
if (0 < sms.items.length) {
    JavaSpace space = (JavaSpace) sms.items[0].service;
    // do something with the space
}
else {
    System.out.println("No Java Space found.");
}
```

Using the `ServiceDiscoveryManager` is similar;

```
ServiceItem si = sdm.lookup(template, null);
if (null != si) {
    JavaSpace space = (JavaSpace) sms.service;
    // do something with the space
}
else {
    System.out.println("No Java Space found.");
}
```

Starting a Transaction Service

What? Wait, why do I need to do that?

Because you'd like to find something that will give you a nice distributed transaction.

- Change to directory to `$RIVER_HOME/examples/hello/`
- Execute the script; `scripts/jrmp-mahalo-group`

Testing it

Testing it is done in the same way as for the Java Space. However, the constructor of the `ServiceTemplate` changes.

```
ServiceTemplate template = new ServiceTemplate(null, new Class[] { TransactionManager.class }, new Entry[0](0.html));
```