Start running Jtrix:
A practical guide

March 2002, for release 0.14.4 onwards

$Id: start-running-jtrix.lyx,v 1.53 2002/01/23 17:40:58 nik Exp $

Nik Silver
feedback@jtrix.org
Jtrix Ltd
57–59 Neal Street, London WC2H 9PJ, UK
+44 20 7395 4990

Contents

1 Introduction

This document does:

- Relate only to release 0.14.4 of the demo package(s) and anything later in the 0.14 series.
- Explain what's going on in Jtrix from a practical perspective.
- Explain how to run your own Jtrix nodes.
- Explain how to run the Jtrix demos.

This document does not:

- Explain what Jtrix is or the technical terms it uses. For this, you are stongly advised to read *Jtrix: How to write netlets* from `http://www.jtrix.org` which introduces things much more slowly. See also the papers *Jtrix: A technical overview*, *Platform specification* or *Jtrix: An introduction for everyone*. You can use *The Jtrix dictionary* as a quick reference.
- Explain installation and compilation. See the packages themselves for that.
- Explain how to write Jtrix applications. See the papers on the site for that *Programming with Jtrix: The Beatrix application framework* and for lower level details *Jtrix: How to write netlets*.

We describe several ways to play with Jtrix:

1. Use a simple Jtrix service. (Section 2.)
2. Run your own nodes with a hosting service. (Section 3.)
3. Launch and run the skeleton application on a remote hosting service. (Section 4.)

   The skeleton application is a simple distributed, self-redundant message service. Its code is discussed in detail in *Programming with Jtrix: The Beatrix application framework*, available at `http://www.jtrix.org/`.
4. Run your own DNS server, HTTP server and servlet engine as Jtrix services. (Section 5.)

   The servlet engine Webtrix is based on the Tomcat Web server at `http://jakarta.apache.org/tomcat`. It is Tomcat set in the

Jtrix environment, meaning that it is a full-spec servlet engine which can be configured to be dynamically scalable and redundant and it runs in conjunction with the DNS server and HTTP server. You can run several such servers together, add more at will, and as you kill them arbitrarily the others will take over. We have taken pains not to alter the original Tomcat servlet engine code at all.

1.1 Notes for Linux users

The examples in this document are shown for Linux. We assume the Jtrix binary directory (the one with all the JAR files and scripts) is on our system path. If not, then this is easily fixed. Assuming the Jtrix binary directory is */usr/lib/jtrix* we just enter this line in the bash shell:

```
export PATH=$PATH:/usr/lib/jtrix
```

Those with a Debian or RPM distribution will find jnode and launcher on their *PATH* anyway.

Please also make sure your *JAVA_HOME* is set properly; please consult your particular Java distribution if you are unsure about this.

Finally, note that the version of Jtrix discussed here is not necessarily compatible with previous versions, so please check you have the latest version. In particular, if you are interacting with a service outside your local environment, such as one at `http://www.jtrix.org`, then different versions are not guaranteed to be compatible. Jtrix.org will always be running the latest version. This will continue until a 1.0 release, although we will of course try to minimise this as much as possible.

If you have any problems, check out the troubleshooting tips in Appendix A or drop us a note via the Sourceforge mailing lists. See at `http://sourceforge.net/projects/jtrix`.

1.2 Notes for Windows users

The examples in this document are shown for Linux, so if you are running Windows® please make the following changes as you read what follows:[1]

[1]Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries.

- Replace forward-slashes with backslashes for directory separators.
- Make sure the Jtrix binaries (the .bat scripts and JAR files) are on your path. This is achieved in either of two ways.

  The first option is to ensure you run everything from within the Jtrix binary directory.

  The second option is to add the Jtrix binary directory to your *Path* environment variable. Let us suppose this is the directory *C:\jtrix\bin*. The we go to Start | Control Panel | System | Advanced | Environment Variables. In the section marked "System variables" select the variable Path and Edit... it so that *C:\jtrix\bin* is on the end. Don't forget to put a *;* (semi-colon) separator in front of it.

  Whichever method you use, you should not put the *.bat* on the end of the script names.
- Make sure your *JAVA_HOME* is set properly. Your particular Java distribution should have done this for you.
- In the long lines that you see below we have added a backslash before each carriage return to tell Linux that we've not finished the line yet. In Windows you will just have to type it all on one long line and then hit carriage return at the end.
- If a Jtrix script or JAR has a parameter of the form *key=value* then you will need to include those in double-quotes like this:

```
C:\> jnode 100 -netlet-stdio hos01-boot.xml "hosting-out=hos01-admin.xml"
     telnet-2000.xml "hosting-in=hos01-admin.xml"
     sas01-boot.xml "hosting-in=hos01-admin.xml"
         "hosting-out=hos01-real-admin.xml" "sas-out=sas01-warrant.xml"
         "sas-desc=sas01-real-desc.xml"
```

Finally, note that the version of Jtrix discussed here is not necessarily compatible with previous versions, so please check you have the latest version. In particular, if you are interacting with a service outside your local environment, such as one at `http://www.jtrix.org`, then different versions are not guaranteed to be compatible. Jtrix.org will always be running the latest version. This will continue until a 1.0 release, although we will of course try to minimise this as much as possible.

If you have any problems, check out the troubleshooting tips in Appendix A or drop us a note on the Sourceforge mailing lists.

2 Using a simple service

Note: *At the time of writing the external "hello world" service is not available. However, this section has been left in for demonstration.*

How to connect to a simple Jtrix service. It's a distributed, self-redundant, hello world service accessed via a command line tool called the launcher.

Some people have found problems with this example if they are working from behind a firewall, and if so then all such examples will exhibit the same problems. See Appendix A to test if your system is problematic and for ideas for solutions.

2.1 What is the launcher?

Note: *At the time of writing the external "hello world" service is not available. However, this section has been left in for demonstration.*

The launcher is a command line tool to launch and manage netlets remotely. If you're familiar with Linux command line shells or DOS shells then you'll be comfortable with the launcher. In fact it shares much in common with Linux shells including simple variables and backticks for in-line evaluation.

But the launcher is not just a command line plus some networking features. It's actually got a full Jtrix node embedded in it. This is how it does its work. Its node run access point netlets to talk to the remote applications being used and managed.

Here, we're just going to use the launcher to access a service. It's the hello world service run by Jtrix.org. Later on in Section 4 we'll also use it to launch our own version of this application into a remote hosting service.

If you want to know more about the launcher and its syntax, have a look at Appendix B which explains all about it.

2.2 What to do

Note: *At the time of writing the external "hello world" service is not available. However, this section has been left in for demonstration.*

The service we're going to use is a real on-line service, and just like all Jtrix services it needs a warrant. Go to `http://www.jtrix.org/warrant/hello-warrant.xml` and get a hello warrant for the hello world service. Each warrant is individually named, so we need to have registered first. Save that warrant in a file called *hello-warrant.xml*.

Now we can use the launcher as follows:

```
% launcher
Initialising Nodality...
launcher> connect myservice hello-warrant.xml
launcher> list
myservice
launcher> list myservice
  Facet: org.jtrix.facets1.service.common.IConsoleFacet
  Facet: org.jtrix.project.helloworld.IHelloFacet
  Facet: org.jtrix.project.skeleton3.facets.ISkeletonFacet
myservice.getMessage - ( Return: java.lang.String )
launcher> myservice.getMessage
Hello, world, and thank you for using our service
launcher> quit

%
```

That's it! We've got the message. We've used an arbitrary service, fulfilled by mobile code (a netlet) which has downloaded and linked back to its (distributed) base to do its work, and all in a completely safe, sandboxed environment. Here's a bit more detail. . .

2.3 Discussion

Note: *At the time of writing the external "hello world" service is not available. However, this section has been left in for demonstration.*

The *connect* command takes a warrant and uses it to connect to the service. It all takes place in the background, but the embedded node downloads an access point netlet, which is the launcher's link to the service. The *list* command lists what applications we're connected to. In this case, just the one, which has a name of our choice. Listing a particular application tells us what commands (and facets) it has.

We can see it has a command called *getMessage*, so we issue that command for that application. The request goes back to the main worker netlets, distributed in a redundant cluster of their own, and one of them delivers the message which the launcher returns to us.

The launcher is covered in more detail in Appendix B.

3 Running our own nodes

How to run one node with a hosting service, play with it, and then add other nodes to it. Various options are described, and each is discussed in its own section.

3.1 What we're going to do

We're going to run our own hosting service, plus a couple of other services: telnet and SAS.

If we run a hosting service it means we're running a platform onto which others can launch and run their own applications. This is totally secure. For one thing, they can't do this unless you have created a warrant and given it to them. Also, you can chose what resources they can access, if any.

But more to the point this allows us to run our own applications from remote locations on our own network, and it gives us exactly the same set-up that Jtrix.org is running publicly. So you have a choice of hosting services: Jtrix.org, ourselves, and anyone else who gives us a warrant for their own service.

Before long Jtrix.org will be running an indexing service which allows people to select and use others' hosting services, so they can find the best places to run their applications. All this is fully accountable.

The telnet application we're also launching simply gives us telnet access into our hosting service, so we can manage it relatively easily. We could use the launcher, too.

The SAS application means we have our own storage system for JAR files and netlet descriptors, which generally makes things run smoother. Again, Jtrix.org runs a public SAS if we want to use that.

3.2 Running our first node

What we're going to do here is prepare three Jtrix applications and then run them on our own node. The preparation is creating netlet descriptors which are digitally signed with our own unique key pairs.

Here are the commands to start this from scratch, with the system's responses. First we set things up. See Section 3.5.1 for an explanation of this:

```
% hospitality_init hos01
generating hospitality for hos01...
generating key pair for hos01...
% ls hos01*
hos01-boot.xml hos01.prv  hos01.pub
% telnet_init 2000
configuring telnet on 2000...
% ls telnet-*
telnet-2000.xml
% sas2_init sas01
generating key pair for sas01...
creating sas01 launch descriptor...
configuring sas01...
```

```
% ls sas01*
sas01-boot.xml sas01.prv  sas01.pub  sas01.xml
%
```

Note that Hospitality is the name of Jtrix.org's implementation of a hosting service. So *hospitality_init* initialises the hosting service.

Now we can run the first node with a hosting service. Section 3.5.2 has an explanation of this:

```
% jnode 100 -netlet-stdio hos01-boot.xml hosting-out=hos01-admin.xml \
     telnet-2000.xml hosting-in=hos01-admin.xml \
     sas01-boot.xml hosting-in=hos01-admin.xml \
         hosting-out=hos01-real-admin.xml sas-out=sas01-warrant.xml \
         sas-desc=sas01-real-desc.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting hos01-boot
netlet:100.0.1: This node: 0a77088b0204000001000000

-----------------------------------------------------
GMS: address is victoria:1342
-----------------------------------------------------
netlet:100.0.1: master changed;false:false
netlet:100.0.1: master changed;true:true
netlet:100.0.1: connecting to master: 0a77088b0204000001000000
no mapping for sas-in: SAS Warrant
writing hos01-admin.xml for hosting-out: Admin Warrant
deleting /var/tmp/jtrix/disk-100/dsk-6613601-0
starting telnet-2000
reading hos01-admin.xml for hosting-in: Telnet Hosting Warrant
starting sas01-boot
reading hos01-admin.xml for hosting-in: hosting admin warrant
writing sas01-real-desc.xml for sas-desc: sas descriptor
writing sas01-warrant.xml for sas-out: real SAS warrant
netlet:100.0.1: connecting to SAS
writing hos01-real-admin.xml for hosting-out: real hosting warrant
Bootstrap complete
```

At the end the prompt does not return, since the node is running. Notice the lines that begin *netlet:XXX.0.1*. That *XXX* is the node ID we chose on the command line. We should make a note of it for later.

Also, make sure you leave the node running in its own window; don't "background" it. It will continue to produce all kinds of output, and you don't want that to interfere with any other things we're going to.

Starting the node with these applications also produced some more files. Here is everything we have so far:

```
% ls hos01* telnet-* sas01*
hos01-admin.xml       hos01.prv       sas01-real-desc.xml  sas01.pub
hos01-boot.xml        hos01.pub       sas01-warrant.xml    sas01.xml
hos01-real-admin.xml  sas01-boot.xml  sas01.prv            telnet-2000.xml
%
```

3.3 Playing with our hosting service with telnet

We have chosen to start our hosting service with a telnet server for administration. We can telnet in and have a little play with it. Our interaction is minimal, and some of the output is large, but it's a start.

See Section 3.6 for a longer discussion of this, but this is what we're going to do. First we connect to the hosting service via telnet. You'll probably have to use your fully qualified domain name or IP address rather than just "localhost".

After connecting we'll get a list of commands. Then we'll have a look at hosting service's status, much of which is meaningless to anyone but the authors of the applications on the node. However, it shows broadly that things are running. Then we'll create a new hosting contract, which is an account allowing someone to run netlets on our hosting service and we'll take "control" of this account, meaning that we'll switch into it to see what's going on. After seeing what commands we have at our disposal, and seeing the status of that contract, we'll quit out of that contract and quit telnet.

```
% telnet victoria.intranet.hyperlink.com 2000
Trying 10.119.8.139...
Connected to victoria.intranet.hyperlink.com.
Escape character is '^]'.
admin> help
available commands:
Help - show this help
Quit - Close connection
status [<oid>] - show global hospitality status
control <contractid> - work with a specific contract
warrant <contractid> - get a contract control warrant
create [<oid> <string>]* - create a contract with arbitrary extra information
adminwarrant - get an admin warrant
list - list contract IDs
kill [<contractid>] - kill a contract
admin> list
Contracts
=========
contractid: 0
contractid: 1
admin> status
10.100.0: 100
10.100.3.6613600.1.0: org.jtrix.facets1.node.ISocketFactory
10.100.3.6613600.2: true
10.100.3.6613600.3: true
10.100.3.6613600.4.102.3: 100
10.100.3.6613601.1.0: org.jtrix.facets1.util.io.IFileSystem
10.100.3.6613601.2: true
10.100.3.6613601.3: true
10.100.3.6613601.4.101.1: true
10.100.3.6613601.4.101.2: true
10.100.4.1: 7
10.100.4.2: 4485688
10.100.4.3: 17371128
30.0.1: 100
30.0.2: 0
30.0.3: 0
30.0.4: org.jtrix.project.beatrix.plugins.netlets.Manager
```

```
30.0.5.0: 0
30.0.5.1: 0
30.0.5.2.0: 0
30.0.5.2.1: 0
30.0.5.2.2: 0
30.0.5.3: 0
30.0.6.1.1: <<null>>
30.0.6.1.2: Beatrix Netlet: O=jtrix, CN=sas2, N=sas01, PID=8907
30.0.6.2.0: Wed Dec 05 10:51:53 GMT 2001
30.0.6.2.1: 0a77088b0404000001000000
30.0.6.2.2: Initial Manager
30.0.6.2.3.0: LEADER
30.0.6.2.4.0: admin
30.0.6.2.4.1: admin-embedded
30.0.6.2.4.2: contract
30.0.6.2.4.3: contract-embedded
30.0.6.2.8: O=jtrix, CN=sas2, N=sas01, PID=8907
30.0.6.2.9: 2
30.0.6.2.10: 60
30.0.7: 0
30.1.1: 100
30.1.2: 0
30.1.3: 0.-1767796489
30.1.4: org.jtrix.project.beatrix.plugins.netlets.Worker
30.1.5.0: 0
30.1.5.1: 0
30.1.5.2.0: 0
30.1.5.2.1: 0
30.1.5.2.2: 0
30.1.5.3: 0
30.1.6.1.1: <<null>>
30.1.6.1.2: Beatrix Netlet: O=jtrix, CN=sas2, N=sas01, PID=8907
30.1.6.2.0: Wed Dec 05 10:52:03 GMT 2001
30.1.6.2.1: 0a77088b0504000001000000
30.1.6.2.2: worker_pool_1_1007549515659.0
30.1.6.2.3.0: WORKER
30.1.6.2.3.1: worker_pool_1
30.1.6.2.5: 0
30.1.6.2.7.45632.0: victoria.intranet.hyperlink.com/10.119.8.139
30.1.6.2.7.45632.1: 50000
30.1.6.2.8: O=jtrix, CN=sas2, N=sas01, PID=8907
30.1.7: 1
50.0.2.0.1: 6613601
50.0.2.1.1: 6613600
50.0.3: 0
50.1.3: 1
90.0.1: 6613601
90.0.2: 0
90.0.4: 100
90.1.1: 6613600
90.1.2: 0
90.1.4: 100
100.10.1: <<warrant>>
100.10.2: <<byte array>>
100.10.3: 2
100.10.4: 2
100.10.5: 2
100.10.7.6613600.10.1: 50000
100.10.7.6613600.20.50000: true
100.10.7.6613601.10.0: dsk-6613601-0
100.10.8.6613600.50000: 100
100.10.8.6613601.10.0: 100
100.20.1.20.0: console created contract
100.20.1.20.1: sample
100.30.100.1: 20
100.30.100.2: 0a77088b0204000001000000
admin> create
created contract #2
```

```
admin> list
Contracts
=========
contractid: 0
contractid: 1
contractid: 2
admin> control 2
control> help
available commands:
filter [<event>]* - set auditing filter
connectresource <node> <instance> - create a resource
setmanager <warrant url> <time> - Set application manager and lapse time
exec <node> <descriptor url> [<name> <id>]* - execute a netlet
dmesg [-c] - fetch (and flush) auditing buffer
Help - show this help
Quit - Close connection
status - show hospitality status for this contract
disconnectresource <instance> - disconnect a resource
releaseresource <instance> - create a resource
kill <netlet id> - kill a netlet
createresource <type> - create a resource
cancellease - release management lease
acquirelease <wait> <time> - wait <wait> ms for management lease of <time> ms
control> status
10.100.0: 100
10.100.3.6613600.1.0: org.jtrix.facets1.node.ISocketFactory
10.100.3.6613600.2: true
10.100.3.6613600.3: true
10.100.3.6613600.4.102.3: 100
10.100.3.6613601.1.0: org.jtrix.facets1.util.io.IFileSystem
10.100.3.6613601.2: true
10.100.3.6613601.3: true
10.100.3.6613601.4.101.1: true
10.100.3.6613601.4.101.2: true
10.100.4.1: 7
10.100.4.2: 4485688
10.100.4.3: 17371128
50.2.3: 2
control> quit

admin> quit

Connection closed by foreign host.
%
```

3.4 Starting a second node

Our second node can now be started anywhere on the same LAN. This time it's much simpler, but the steps are slightly different to what we've seen before, including giving this jnode a different node ID:

```
% telnet_init 2001
configuring telnet on 2001...
% ls telnet-*
telnet-2000.xml  telnet-2001.xml
% jnode 101 -netlet-stdio hos01-boot.xml hosting-out=hos01-admin.xml \
     telnet-2001.xml hosting-in=hos01-admin.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting hos01-boot
netlet:101.0.1: This node: 0a77088b0704000001000000
```

```
-----------------------------------------------------
GMS: address is victoria:1292
-----------------------------------------------------
netlet:101.0.1: master changed;true:false
netlet:101.0.1: connecting to master: 0a77088b0204000001000000
no mapping for sas-in: SAS Warrant
writing hos01-admin.xml for hosting-out: Admin Warrant
starting telnet-2001
reading hos01-admin.xml for hosting-in: Telnet Hosting Warrant
Bootstrap complete
```

There's only one setup script, but we must reuse the previous descriptor of Hospitality boot netlet. This is discussed in Section 3.7. The hosting service spreads out to this second node and SAS follows it. Also we can telnet to this second node on port 2001.

3.5 Discussion: Starting our node

What did we do just now? Some explanation follows; more information on these commands is contained in the appendices.

3.5.1 Setting up the hosting service

First we generated a key pair and netlet descriptor for a new hosting service called *hos01*. That created the files listed. The XML boot file is a netlet descriptor for the hosting service which allows people to run netlets it. In fact the Hospitality application is a distributed hosting service, and no more. It simply allows several nodes to work together to provide a service, and that service is netlet hosting. The other files are a public/private key pair to secure the hosting service. It allows us to ensure our Hostpitality's future warrants and descriptors (i.e. the hosting service's future warrants and descriptors) are signed.

Next we generated another descriptor. This for a descriptor which provides telnet access into the hosting service for admin purposes. We have specified that the access should be on port 2000. This is a very small and simple application so it doesn't bother with keys.

The final step in the setup is to generate SAS files and corresponding keys. We have labelled our SAS application *sas01*. You might want to change it to for own name.

The XML boot file is a netlet descriptor allowing us to run the SAS, and of course we have the key pair. The file *sas01.xml* is a launch descriptor, which allows us to start off SAS via the command line launcher application.

If we look at the netlet descriptors we find they are very large; this is because they contain all their own JARs serialised into them. It would

be better to reference them by URL and download them, but at that stage in the proceedings there is nowhere to do that from. On the other hand, the SAS launch descriptor is quite small. It refers to its JARs as files on the local file system.

Why don't the netlet descriptors contain local file references? Because they need to run inside nodes, and nodes could be anywhere in the world; they don't usually run on our local machine. So why can the launch descriptor use local file references? Because it is read by the launcher console which runs on a local machine and which turns it into a netlet descriptor with no local file references.

3.5.2 Running the first node

The *jnode* script (Appendix D) starts a new node. Jnode is just a runnable node which also has group communication features. It is thus little more than a runnable version of Nodality. See Appendix C.

The first *jnode* parameter is a node ID. We can use any number, as long as this is unique for each node in the LAN. The second parameter just tells the node to output the standard I/O of all its netlets. Useful for debugging and general interest.

The rest of the parameters are just a series of netlet descriptors, each of which is followed by a number of netlet-specific arguments of the form *X*=*Y*.

Each of these netlets is a *bootstrap netlet*, which is just a netlet which will have special access to local I/O streams. This stream access is a special privilege because netlets are not normally granted direct access to system resources. However jnode gives this privilege only to those netlets we've specified on the command line, and initialises those netlets in sequence. Each *X*=*Y* argument is one of these I/O streams (*Y*) with a name (*X*) which the netlet is expecting.

For example, we can see from the output above that the hos01-boot netlet (from the netlet descriptor *hos01-boot.xml*) outputs to its stream called *hosting-out*. It writes an admin warrant to file *hos01-admin.xml*, which we specified on the command line. We can also see, on the line before, that it tried to read a SAS warrant from the stream called sas-in, but since we've not specified this it does without.

Also notice how the telnet netlet starts, next. It reads the file *hos01-admin.xml* from its stream called *hosting-in*, which is a file just output from the previous netlet. Similarly for the SAS netlet, which reads from one stream and writes to three others.

Stringing netlets together in this way is only possible when booting Jnode.

Finally, the files produced by the boot process:

- *hos01-admin.xml* is a warrant to use the hosting service just started. It is an admin-level warrant and is used by both the telnet and SAS netlets. It is actually only usable on the current node because a warrant should point to a SAS and we don't have a SAS at this stage. So this warrant is, and can only be, used by the other bootstrap netlets on this node.
- *sas01-real-desc.xml* is a descriptor for our SAS. Although we already have a SAS descriptor that contained the JARs embedded in it, and hence was very large. But by this stage the SAS has loaded its own JARs into itself. Hence this "real" descriptor references the now-downloadable JARs as URLs.
- *sas01-warrant.xml* is a warrant for admin-level access to the SAS.
- *hos01-real-admin.xml* is another admin-level warrant for the hosting service. Unlike the other warrant this is usable from any node because the hosting service's JARs are now uploaded into the SAS.

3.6 Discussion: Playing with our node with telnet

Telnet access is to administrate the hosting service, and nothing more.

The list command lists the current contracts. The status command gives a current view of the entire hosting service state. Each status line shows a dotted object ID (*oid*) and some value. This approach has been inspired by SNMP. For a description of the oids used by the hosting service, see the class `org.jtrix.project.cluster.facet.Oids`.

In the example above we create a new contract which is given an ID, and then issue a command to manage that contract. In fact all we do is check the status of that particular contract. Then we quit the contract control command line and quit the telnet interface.

Other commands can be found with the *help* command, and we use some of these in the hello world examples.

3.7 Discussion: Starting a second node

Starting the second node in the hosting service was different to the first. Why did we only need an initialisation script for telnet? What about SAS? Don't we want to put that on this new node, too? And why did we reuse the Hospitality boot descriptor? The answers, in

short, are because they are different applications of varying levels of sophistication. . .

The Hospitality boot descriptor has to be reused. It includes an identifier, *hos01*, for this particular hosting service. That same netlet descriptor executes the same netlet, which, when it initialises, will scour the LAN for another hosting netlet with the same identifier. It finds the original and they join together.

The telnet netlet is very simple-minded. All it does it listen on a given port and provides command line access to the hosting service. It doesn't have the intelligence to distribute itself. Also, assuming this second node is on the same machine as the first, they couldn't both listen on the same port, so it does need to be given its port number.

SAS, on the other hand, is very sophisticated. It is built with Beatrix which makes good use of the hosting service. So as soon as the hosting service is running on two nodes Beatrix will realise this and spread SAS out across both of them.

Starting our second node raises some other interesting issues. For example, SAS delivers netlet descriptors and JAR files via URLs. Both SAS netlets do this. Are they listening on different ports? If they're on the same machine then yes, they must be. Does this mean the descriptors and JARs are available at different URLs? Yes, it does. And a warrant references a service, which is really a location it can get netlet descriptors and JARs. So how does a warrant cope with multiple locations like this? Well, it simply contains several alternative URLs. . .

But suppose we started a SAS on one node, uploaded a service to it, and then made a warrant referencing the service in this SAS. It would only contain one URL, right? Right. And suppose we the started a second node and SAS spread itself out to that. What would our warrant say then? Well, a warrant is an XML file; it doesn't change magically just because some server's started up somewhere in the world. It still references the same one node, the first one. Okay, so what happens if the first node shuts down? By the magic of Jtrix SAS is still running on the second node, but isn't our warrant now out of date, since it points only to the first node? Yes, the warrant is out of date. So isn't that a problem? Well, it's a general problem of the Internet—if you rely on a service from one location and that location disappears then can't access the service—but, yes, that warrant is now useless. So what can we do about that? There are several things. . .

From the warrant-holder's point of view they should update their warrant from time to time. When they are connected to the service as a consumer they may do all their consumer tasks and also say "Please give me an updated warrant". The service, if it supports such a function, can give them a warrant with all they lastest SAS URLs. Beatrix helps applications support this function very easily.

The warrant holder, if they still find themselves with a useless warrant, could contact the service administrator directly and ask for an update. Of course this could be very inconvenient. At worst it would mean phoning them up, remembering their contract number and providing proof of who they are.

The real onus is on the service provider. For one thing they shouldn't give out warrants if they know their service only lives on one server which is likely to go down. They should pick their hosting service with care. This is helped further if they spread their service over several hosting services, which is exactly the kind of thing that Jtrix is designed for.

Second, this problem really only occurs if the single SAS URL is specified by IP address. If it's specified by a fully qualified host name then the DNS can be repointed. An even smoother version of this is possible when the DNS service is itself a Jtrix service, as is used in the Webtrix example below. If SAS is a consumer of the Jtrix DNS service then as soon as it notices its first server has gone down it can repoint the host name to the second server. The warrant-holder will never know the difference.

3.8 Creating our own warrants

The skeleton example below uses hosting and SAS warrants downloaded from Jtrix.org. But if we have our own hosting service then we can create our own warrants and use those. Here's how we do that through the launcher console.

Roughly this is what we're going to do. First we connect to our hosting service as its administrator, and get make a new contract and warrant for someone to run their netlets on our hosting service. Then we connect to SAS as its administrator and do a similar thing, creating a contract and warrant for a consumer to store JARs and netlet descriptors. Along the way we dump these warrants into files and see what other commands we might issue as users of these services. Please do expect pauses which some code is downloaded to be run (securely) in the launcher.

```
% launcher
Initialising Nodality...
launcher> connect hostadmin {warrant:hos01-real-admin.xml}
launcher> contract=‘hostadmin.createContract sample‘
launcher> echo $contract
1
launcher> hw=‘hostadmin.getWarrant $contract‘
launcher> echo $hw
<<warrant>>
launcher> dump $hw hosting-warrant.xml
launcher> disconnect hostadmin
```

```
launcher> connect sasadmin {warrant:sas01-warrant.xml}
launcher> list sasadmin
  Facet: org.jtrix.facets1.service.common.IConsoleFacet
  Facet: org.jtrix.facets1.service.common.IWarrantFacet
  Facet: org.jtrix.project.sas2.facet.ISASAdmin
sasadmin.add-hosting - <id> <warrant> - Add a hosting service to this application
sasadmin.createContract - ( Arguments: org.jtrix.base.X500DN boolean Return: org.jtrix.
base.Warrant )
sasadmin.deleteContract - ( Arguments: org.jtrix.base.X500DN Return: void )
sasadmin.fetchContract - ( Arguments: org.jtrix.base.X500DN boolean Return: org.jtrix.
base.Warrant )
sasadmin.get-warrant - <type> [<service-data>] Get refreshed warrant
sasadmin.list-hosting - Display the hosting services
sasadmin.list-management-state - Display the management state
sasadmin.list-managers - Show all managers and their nodes
sasadmin.list-workers - Show all workers and their nodes
sasadmin.listContracts - ( Return: List of org.jtrix.base.X500DN )
sasadmin.quit - Shut down the application
sasadmin.refresh - ( Return: org.jtrix.base.Warrant )
sasadmin.set-manager-redundancy - <min> <max> Set the manager redundancy
launcher> sasadmin.listContracts
N=hos01
O=jtrix, CN=sas2, N=sas01, PID=9433
launcher> sasadmin.createContract {x500dn:uid=nik,o=jtrix.org} {boolean:false}
<<warrant>>
launcher> sw=$0
launcher> dump $sw sas-warrant.xml
launcher> sasadmin.listContracts
N=hos01
O=jtrix, CN=sas2, N=sas01, PID=9433
UID=nik, O=jtrix.org
launcher> disconnect sasadmin
launcher> quit

%
```

The two dump commands have created two new warrants. These can now be used to run the skeleton application, or indeed any other.

3.9 Discussion: Creating our warrants

The session above shows:

1. As a hosting administrator we connect to the hosting service (the application called Hospitality). Then we create a new contract for a customer and create a hosting control warrant for them in variable *$hw*. This warrant is a consumer-level warrant and gives the holder the rights to run an application on the hosting service.
2. We we write the warrant to a file *hosting-warrant.xml* and disconnect from the hosting admin service.
3. We do the same thing for SAS: connect with our admin-level warrant, create a new customer contract, and put their consumer-level warrant into variable *$sw*.

4. We dump the warrant to *sas-warrant.xml* and disconnect from the SAS admin service. Finally we quit the launcher.

Notice that although both the hosting service and SAS have the idea of contracts, the conepts are different and hence hence so are the command line interfaces. For the hosting service a new contract is an integer (which we can see when we issue the command *echo $contract*), and from this we can get a consumer warrant. But for the SAS a contract is identified by an X.500 DN, and creating a new contract does itself create a new warrant.

If we were to now telnet into our hosting service and use its *list* command we would see the new hosting contract.

4 Running the skeleton application

What follows is based on the skeleton application described in *Programming with Jtrix: The Beatrix application framework* at `http://www.jtrix.org`.

You may need to change some of what follows according to where your Jtrix JARs are located.

4.1 Preparing the application

Before we begin we need to create a launch descriptor, which we will name *skeleton-launcher.xml*. We do it like this. As usual alter the Jtrix binaries directory to how it on your system. We're using */usr/lib/jtrix*:

```
% jtrixmaker -type beatrix -outfile skeleton-launcher.xml \
     -x500dn o=jtrix,cn=skeleton2 -jardirs /usr/lib/jtrix \
     -jars skeleton2.jar beatrix.jar libjtrix.jar jaxp.jar parser.jar facets1.jar \
     -access ap \
     -codebase ap skeleton2.jar beatrix.jar libjtrix.jar facets1.jar \
     -plugins org.jtrix.project.skeleton2.plugins.LifeCyclePlugin
%
```

First we say what type of descriptor we're generating (a Beatrix launch descriptor) and what it should be called. Then there is an X.500 distinguished name (DN) which you might like to change according to your own circumstances, e.g. *O=MyCompany,UID=BobJones,CN=skeleton*. The other arguments are specific to this application. See Appendix F for more on the jtrixmaker command.

Now we have a launch descriptor called *skeleton-launcher.xml*.

Before we can launch it we need a hosting warrant (giving us permission to run an application) and a SAS warrant (which allows us to upload and download netlet descriptors, and hence make things generally more convenient. We will assume these are in files *hosting-warrant.xml* and *sas-warrant.xml*. See the sections above on how to do this.

4.2 Launching and running the application

Here is how we can do this.

First we'll start the launcher and announce which hosting service and SAS we're going to use:

```
% launcher
Initialising Nodality...
launcher> host {warrant:hosting-warrant.xml}
launcher> sas {warrant:sas-warrant.xml}
SAS warrant recognised: will upload jars
launcher>
```

Next we launch the skeleton application. This will upload various JARs, start the application, and download an access point netlet, all of which will take some time:

```
launcher> run sk skeleton-launcher.xml
Uploading skeleton2                          [5....4....3....2....1....0]
Uploading beatrix                            [5....4....3....2....1....0]
Uploading parser.jar                         [5....4....3....2....1....0]
Uploading libjtrix                           [5....4....3....2....1....0]
Uploading jaxp.jar                           [5....4....3....2....1....0]
Uploading facets1.jar                        [5....4....3....2....1....0]
Waiting for application to initialise... done
launcher>
```

The skeleton application has be written with debugging switched on, so you will see lots of debug output in your node window after you launch it.

When the prompt returns we will have become the application's administrator. Then we dump an admin warrant (so we can connect to it again in future). After that we generate a basic consumer-level warrant, connect as a consumer and use the service (i.e. we get the message). Finally, before quitting, we dump the consumer warrant to file, just in case we want to connect to the service in future, or perhaps mail it to a friend. Here's what that all looks like:

```
launcher> list
sk
launcher> list sk
  Facet: org.jtrix.facets1.service.common.IConsoleFacet
sk.quit - Shut down the application
```

```
sk.list-managers - Show all managers and their nodes
sk.list-workers - Show all workers and their nodes
sk.get-warrant - <type> [<service-data>] Get refreshed warrant
sk.set-manager-redundancy - <min> <max> Set the manager redundancy
sk.list-management-state - Display the management state
sk.add-hosting - <id> <warrant> - Add a hosting service to this application
sk.list-hosting - Display the hosting services
launcher> dump $sk skadmin.xml
launcher> bw=‘sk.get-warrant skeleton‘
launcher> connect skbasic $bw
launcher> list
skbasic
sk
launcher> list skbasic
  Facet: org.jtrix.facets1.service.common.IConsoleFacet
  Facet: org.jtrix.project.skeleton2.facets.ISkeletonFacet
skbasic.getMessage - ( Return: java.lang.String )
launcher> skbasic.getMessage
Hello, skeleton plugin world
launcher> echo $skbasic
<<warrant>>
launcher> dump $bw skbasic.xml
launcher> quit

%
```

Even after quitting our warrants will still be on our local file system and usable in case we want to start up the launcher again.

4.3 Discussion: What skeleton does

The skeleton application is a distributed, always-available, self-replicating message service, albeit the same message every time.

It has two levels of access: admin level allows someone to set redundancy levels and perform other such tasks; consumer level only allows access to the message. Each kind of access has two interfaces: command level access, used by the launcher above, and programmatic access, in case we want to administrate it or get its message within another Jtrix application.

It is distributed in the sense that when it starts it spreads itself out over several nodes within the hosting service. If you are running it on your own one-node hosting service, however, you will see exceptions thrown on jnode’s I/O as it fails to find an adequate number of nodes. Even if you have the full four nodes it requires you will see lots of debugging output as the skeleton application has debugging switched on by default.

It is always available in the sense that connecting to it will connect to any of the nodes on which it currently runs.

If one of these nodes dies then it will replicate itself to another node in the same hosting service.

Skeleton is built on the Beatrix application framework, and most of the functionality above is provided automatically by Beatrix, including the self-replication and the admin features.

4.4 Discussion: Launching and running

As we've already mentioned, the above is discussed more in *Programming with Jtrix: The Beatrix application framework* at `http://www.jtrix.org` and Appendix B. But here's a summary.

First we announce the use of whichever hosting service we're going to use. After this point the launcher knows where to launch any applications to.

Next we announce which SAS we're going to use. This is not obligatory, but it's incredibly helpful. It tells the launcher that when we launch an application it can upload the JARs into that SAS, which saves it serializing them into any netlet descriptors and warrants each time it wants to generate one.

Then we run (i.e. launch) the skeleton application on the hosting service. The launcher takes the application descriptor, *skeleton-launcher.xml*, and sees which JARs it needs. It reads them out of the local file system and uploads them into the SAS. Then it creates a netlet descriptor which references those JARs by URL and runs that on the hosting service. The application initialises on the remote node and returns a warrant. At this point we are the application's administrator and have admin-level access to it.

We have labelled the application *sk* and our admin warrant is in variable *$sk*. Listing all our applications confirms we're only connected to that one. Listing the commands in that application gives various options. We dump the admin warrant into file *skadmin.xml*.

One of the commands allows us to get a warrant from the skeleton application, so we create a new basic warrant in variable *$bw*. This gives us access to the basic "skeleton" consumer-level service. The backticks allow us to execute a command and return the result in-line.

We connect with this warrant. Listing all our applications confirms we have two application connections. Listing the commands in the *skbasic* connection shows it presents a different set of commands. This makes sense—consumers have different requirements to administrators. One of the commands allows us to get a message, so we issue that command.

Finally we dump the basic warrant into a file and quit the launcher. Quitting the launcher does not quit the application, however; that

keeps running until told to quit by the *sk.quit* command. Now we can e-mail the basic warrant to others so they too can access our service as consumers, from anywhere in the world.

Notice that we've also saved our admin warrant into its own file. We can use this to reconnect to the application at any time in the future to perform admin work.

Another point to note is that the different levels of service here (admin and consumer, called *skeleton*) are entirely up to the application. Other applications may have any kinds they wish, with any names they wish.

5 Running our own Web server

A much more involved example: buying in a servlet engine to run a Web site.

We start three services—DNS, HTTP and the servlet engine back end, Webtrix—and for each one we create a warrant so some consumer can use them. We then pretend to be that consumer, who wants to run *www.acme.com*. We (a) tell the DNS we want to run a *www* subdomain of *acme.com*; (b) tell the HTTP service we'll need some disk space for our servlet engine back end; (c) upload a WAR file into our servlet engine under the */examples* hierarchy; (d) link the HTTP service to the DNS and Webtrix. Once that's done we can access our site's URLs which start `http://www.acme.com/examples`.

It's worth re-reading that description. The reason we link the HTTP service to the other two is as follows. The HTTP servers need to forward requests to the servlet engine back end, so it must be aware of Webtrix, and if an HTTP server fails the DNS record can be updated to point to a replacement HTTP server.

Refer to the Beatrix documentation for a bit more on how these applications are designed. We'll discuss them as we go along.

Please note that for this example to work you do need the java compiler, javac, to be on your command path. This is because the servlet engine needs it to compile JSPs.

5.1 Starting a suitable node

If we haven't already created suitable descriptors, we need to do this now:

```
% hospitality_init hos01
generating hospitality for hos01...
generating key pair for hos01...
% ls hos01*
hos01-boot.xml hos01.prv  hos01.pub
% telnet_init 2000
configuring telnet on 2000...
% ls telnet-*
telnet-2000.xml
% sas2_init sas01
generating key pair for sas01...
creating sas01 launch descriptor...
configuring sas01...
% ls sas01*
sas01-boot.xml sas01.prv  sas01.pub  sas01.xml
%
```

In this example we need an HTTP server (port 80) and a DNS server (port 53). Such access to privileged ports is not granted to just anyone. We need the following:

1. The nodes must be running as the machine's administrator (e.g. root on Linux, administrator on Windows) or any other user who can start services on those ports.
2. Our node must be told explicitly to make these ports available to netlets.

Therefore we become the administrator and run the following. (**Note for Windows users:** The file jnode.bat does not support the –webres option. Instead, have a look in that file with a text editor. It will tell how to make a simple change to get the same effect.)...

```
# jnode 104 -webres hos01-boot.xml hosting-out=hos01-admin.xml \
     telnet-2000.xml hosting-in=hos01-admin.xml \
     sas01-boot.xml hosting-in=hos01-admin.xml \
         hosting-out=hos01-real-admin.xml sas-out=sas01-warrant.xml \
         sas-desc=sas01-real-desc.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting hos01-boot

-----------------------------------------------------
GMS: address is victoria:1455
-----------------------------------------------------
no mapping for sas-in: SAS Warrant
writing hos01-admin.xml for hosting-out: Admin Warrant
deleting /var/tmp/jtrix/disk-104/dsk-6875745-0
deleting /var/tmp/jtrix/disk-104/dsk-6875745-3
deleting /var/tmp/jtrix/disk-104/dsk-6875745-5
starting telnet-2000
reading hos01-admin.xml for hosting-in: Telnet Hosting Warrant
starting sas01-boot
reading hos01-admin.xml for hosting-in: hosting admin warrant
writing sas01-real-desc.xml for sas-desc: sas descriptor
writing sas01-warrant.xml for sas-out: real SAS warrant
writing hos01-real-admin.xml for hosting-out: real hosting warrant
Bootstrap complete
```

All that's different is the extra argument *–webres* which requests the relevant resources. This really does nothing more than request two port blocks with the *–netblock* option of *jnode.jar*. See Appendix D for the appropriate help file.

5.2 Allowing the services to run

Now we run a launcher session.

This first session connects to the hosting service we just started. We create three hosting contracts (one each for DNS, HTTP and Webtrix) so our applications can run:

```
% launcher
Initialising Nodality...
launcher> connect hosting {warrant:hos01-real-admin.xml}
launcher>
launcher> contractid1=‘hosting.createContract dns-contract‘
launcher> contractid2=‘hosting.createContract http-contract‘
launcher> contractid3=‘hosting.createContract webtrix-contract‘
launcher> echo $contractid1 $contractid2 $contractid3
1 2 3
launcher>
launcher> dns-hosting-warrant=‘hosting.getWarrant $contractid1‘
launcher> echo $dns-hosting-warrant
<<warrant>>
launcher>
launcher> dump $dns-hosting-warrant hosting-warrant1.xml
launcher> dump ‘hosting.getWarrant $contractid2‘ hosting-warrant2.xml
launcher> dump ‘hosting.getWarrant $contractid3‘ hosting-warrant3.xml
launcher>
```

This uses our warrant *hos01-real-admin.xml* from the node boot-up process. We create a contract for each service and a warrant for each contract, then we dump each one into a file. Then we could mail, for example, *hosting-warrant1.xml* to the DNS administrator so they can launch their service on our node.

Next we connect to our SAS and create a SAS contract for our service administrators:

```
launcher> connect sas {warrant:sas01-warrant.xml}
launcher> sas.createContract {x500dn:uid=nik} {boolean:false}
<<warrant>>
launcher> dump $0 sas-warrant.xml
launcher> quit

%
```

This uses *sas01-warrant.xml* from the node boot-up process, and the SAS consumer warrant we create is put into a file. All our services will use this consumer warrant, but this is just a shortcut because we know they're owned by the same person.

5.3 Launching the services

Next we pretend to be the owner of the applications.

First we copy an example WAR file into our current directory (from the Jtrix binary directory) and create launch descriptors:

```
% cp /usr/lib/jtrix/examples.war .
% dns_init nikdns
% http_init nikhttp
configuring http listener...
% webtrix_init nikwebtrix
configuring tomcat backend..
%
```

As mentioned above, a launch descriptor is an XML file used by the command line launcher application. It allows the launcher to start an application running on a remote node, with all the SAS uploads etc taken care of. The launcher also allows general command line access to running applications.

Let us concentrate on the job of the owner of the DNS application.

We launch our application and then create a contract for our consumer who will manage their own domain, *acme.com*. (Note that the DNS's command *dns.putDomain* can produce a NullPointerException; this appears harmless but is being investigated.)

```
% launcher
Initialising Nodality...
launcher> host {warrant:hosting-warrant1.xml}
launcher> sas {warrant:sas-warrant.xml}
SAS warrant recognised: will upload jars
launcher> run dns-admin dns.xml
Uploading beatrix                              [5....4....3....2....1....0]
Uploading parser.jar                           [5....4....3....2....1....0]
Uploading dns_facets.jar                       [5....4....3....2....1....0]
Uploading dns_test.jar                         [5....4....3....2....1....0]
Uploading libjtrix                             [5....4....3....2....1....0]
Uploading dns.jar                              [5....4....3....2....1....0]
Uploading jaxp.jar                             [5....4....3....2....1....0]
Uploading facets1.jar                          [5....4....3....2....1....0]
Waiting for application to initialise... done
launcher>
launcher> dns-admin.putDomain acme.com
launcher> consumer=feedback@jtrix.org
launcher> dns-admin.createContract acme.com {boolean:false} $consumer AREC CNAME SUB MX
6231210018156006958
launcher> contractid=$0
launcher> consumer-warrant=‘dns-admin.getContractWarrant $contractid‘
launcher> dump $consumer-warrant acme-dns-warrant.xml
launcher>
```

Notice that we dump the consumer's warrant into a file. They'll be able to use this later to connect to our DNS service and manage their domain.

Next we pretend to be the owner of the HTTP application. Again we launch the application, make a new warrant for the consumer, and dump it into a file for them:

```
launcher> host {warrant:hosting-warrant2.xml}
launcher> run http-admin http-service.xml
Uploading facets1.jar         Jar already uploaded with same hash and label
Uploading http_service.jar                    [5....4....3....2....1....0]
Uploading libjtrix            Jar already uploaded with same hash and label
Uploading parser.jar          Jar already uploaded with same hash and label
Uploading beatrix             Jar already uploaded with same hash and label
Uploading http_facet.jar                      [5....4....3....2....1....0]
Uploading dns_facets.jar      Jar already uploaded with same hash and label
Uploading tomcat-3.2.3.jar                    [5....4....3....2....1....0]
Uploading jaxp.jar            Jar already uploaded with same hash and label
Waiting for application to initialise... done
launcher>
launcher> http-admin.createContract {x500dn:o=acme,uid=bob}
<<warrant>>
launcher> dump $0 acme-http-warrant.xml
launcher>
```

Notice the HTTP service has a different way of distinguighing contracts—it uses an X.500 DN.

Next we pretend to be the owner of Webtrix. We launch it and create a consumer warrant, which we dump into a file once again:

```
launcher> host {warrant:hosting-warrant3.xml}
launcher> run webtrix-admin webtrix.xml
Uploading facets1.jar         Jar already uploaded with same hash and label
Uploading cos.jar                             [5....4....3....2....1....0]
Uploading libjtrix            Jar already uploaded with same hash and label
Uploading jasper-3.2.3.jar                    [5....4....3....2....1....0]
Uploading parser.jar          Jar already uploaded with same hash and label
Uploading beatrix             Jar already uploaded with same hash and label
Uploading http_facet.jar      Jar already uploaded with same hash and label
Uploading servlet-3.2.3.jar                   [5....4....3....2....1....0]
Uploading tomcat-3.2.3.jar    Jar already uploaded with same hash and label
Uploading webtrix.jar                         [5....4....3....2....1....0]
Uploading jaxp.jar            Jar already uploaded with same hash and label
Waiting for application to initialise... done
launcher>
launcher> webtrix-admin.createContract {x500dn:o=acme,uid=bob}
<<warrant>>
launcher> dump $0 acme-webtrix-warrant.xml
launcher> quit

%
```

Now that we've launched our three services we quit the launcher.

5.4 Using our services

Our next and final role is as the consumer, the owner of *acme.com* who wants to launch and run their Web site.

Our first step is to start a new launcher and connect to the DNS service. We issue some commands to manage our domain:

```
% launcher
Initialising Nodality...
launcher> connect acme-dns {warrant:acme-dns-warrant.xml}
launcher> acme-dns.addSubdomain www
launcher> webmaster=feedback@jtrix.org
launcher> acme-dns.issueWarrant www $webmaster AREC null null null
<<warrant>>
launcher> www-acme-warrant=$0
launcher>
```

What we're doing here is creating another warrant. This one is for the HTTP service so it can talk to the DNS service. We don't want to give the warrant we just used, because we don't want to let the HTTP service manage our entire domain—we just want it to manage the *www* subdomain. So we've created a warrant which just allows it to manage the A records of that subdomain.

Our next step is to connect to the HTTP service. The details of what we do here are beyond the scope of this document, but briefly we just tell it that we'll need some disk resources:

```
launcher> connect http {warrant:acme-http-warrant.xml}
launcher> file-facet-reqs={!http.string_array:org.jtrix.facets1.util.io.IFileSystem}
launcher> file-properties-reqs={!http.propertyset:}
launcher> resource-reqs={!http.rrequirements:null}
launcher> http.findResources $file-facet-reqs $file-properties-reqs
<<object>>
launcher> resource-reqs=‘http.addrequirements disk $0 $resource-reqs‘
launcher>
```

Next we connect to our Webtrix service and upload our WAR file:

```
launcher> connect webtrix {warrant:acme-webtrix-warrant.xml}
launcher> webtrix.uploadWar examples << examples.war
launcher> webtrix.addBackend /examples examples
launcher> backend=‘webtrix.getBackendWarrant‘
launcher>
```

In uploading the WAR we've also told Webtrix that it should appear under the */examples* hierarchy. We've also created a warrant to access it; this will be given to the HTTP service.

At this stage we've set up our DNS, primed our HTTP server and uploaded our servlet-based Web site.

Here's how we link them all together. The second command will take a long time to execute:

```
launcher> http.setDNSWarrant $www-acme-warrant
launcher> http.setListener $backend www.acme.com $resource-reqs
<<object>>
launcher>
```

That second command takes a long time because a huge amount is going on. Among other things HTTP server needs to connect to Webtrix, which starts up Tomcat, loads the WAR files and starts up the site. Along the way several new netlets are started. But once that's done our Web site is up.

Here's how we find out what IP address *www.acme.com* is listening on (i.e. its A record). This was set in the DNS by the HTTP service, and we are connecting to the DNS service using the same warrant it used:

```
launcher> connect www-dns $www-acme-warrant
launcher> www-dns.getA www.acme.com
10.119.8.139
launcher> quit

%
```

5.5 Testing our Web site

Now we can connect to *www.acme.com* and fetch whatever servlets it has.

Actually, we don't really own *acme.com*, so we can't get to it by host name, and the HTTP service does need to respond to the host name rather than the IP address since, like any good Web server, it expects to run several domains under the same IP address. So the following test is with telnet by IP address, using the `Host:` header to force the right response:

```
% telnet victoria.intranet.hyperlink.com 80
Trying 10.119.8.139...
Connected to victoria.intranet.hyperlink.com.
Escape character is '^]'.
GET /examples/servlets/index.html HTTP/1.0
Host: www.acme.com

HTTP/1.1 200 OK
Content-type: text/html
Content-length: 4516
Last-modified: Wed, 05 Dec 2001 11:36:36 GMT

<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
   ....
Connection closed by foreign host.
%
```

Notice we the document we fetch is prefixed */examples*, as we specified above to Webtrix. The server's copious HTML response is truncated.

Of course, if our workstation was really using our new DNS service, then we could have accessed the site via any Web browser.

A Troubleshooting

Some things you might want to be aware of:

- People working behind firewalls can have problems with the examples which don't run entirely locally. This used to be a problem when Jtrix Ltd ran a public "hello world" service, for example (Section 2). This is no longer the case, but problems can still occur if you are trying to access other external services.

  To see if your firewall is problematic then do two things. First, have a look in *hello-warrant.xml* and look for some bind servers URL, like this:

  ```
  ...
  <bind-server>
    <url>http://213.52.152.66:30100/service/2/skeleton</url>
    <url>http://213.52.152.67:30000/service/2/skeleton</url>
  </bind-server>
  ...
  ```

  There are two URLs. Next, pick any of these and try to telnet to it. If we pick the first one in the above example (address 213.52.152.66, port 30100) we get this on Linux:

  ```
  % telnet 213.52.152.66 30100
  Trying 213.52.152.66...
  Connected to 213.52.152.66.
  Escape character is '^]'.
  ```

  Everything went well in this example—we connect to the server and get a prompt telling us to hit Control-] to exit. But if your system has problems you will not get this far—it will hang or return some error.

  The current solutions are (1) to run such examples from a location without a firewall or with a different firewall, or (2) jump to the other examples which are local only, and do not require external connections. Also, look on `http://sourceforge.net/projects/jtrix` which will have some discussion on this issue.

- Please check you have the latest version of Jtrix. In particular, if you are interacting with a service outside your local environment, then different versions are not guaranteed to be compatible. Jtrix.org will always be running the latest version. This will continue until a 1.0 release, although we will of course try to minimise this as much as possible.

- If you are on a dial-up machine and want to run Jtrix on the Internet then make sure you dial-up before you start Jtrix. Otherwise it can timeout before it gets a connection. However, if you just want to run Jtrix locally then this isn't an issue.

- Some Linux machines may find problems receiving connections from Hospitality. This can happen if the */etc/hosts* file either maps 127.0.0.1 to the machine name, or maps localhost to the machine name. The solution is to change the */etc/hosts* file. Lines that begin "127.0.0.1" or *localhost* should have the machine name removed.

- To run the Hospitality application you will need a default route. If you are on a Linux machine without networking you may not have one, but you can add a default route going to your local machine (127.0.0.1) with this command:

  ```
  /sbin/route add default gw 127.0.0.1
  ```

  Another option (suitable for both Windows and Linux users) is to amend the *jnode* script (or *jnode.bat* on Windows) so that the –shared option is set for that default route.

- When running the skeleton application you may find that binding the basic *skeleton* warrant fails. This may happen if you generate and bind the warrant too quickly; the solution is to wait and try again. The reason is that the *skeleton* service relies on a worker netlet, and these are not created instantly by the worker pool.

- When you shut down and restart an entire node cluster some applications lose their state. One example is SAS. A user's SAS warrant refers to a contract in its database, and this database is (currently) held in memory. So on a restarted cluster the original warrant is found to be invalid and a new one needs to be obtained. On the other hand, the SAS administrator's admin warrant is still valid, and they can use this to reissue the new warrant.

- In older versions of Jtrix, we found some Linux systems ran out of threads quite quickly when running two or more nodes on the same machine. The workarounds are either: run Jtrix across different machines; run different nodes as different users; recompile the kernel allowing more threads per user. For this last option look in *include/linux/tasks.h* and increase *NR_TASKS*; going to 4000 should be fine. We have not found this a problem on current releases, however.

- Some people have got an OutOfMemoryError when running the Webtrix example. This is a silent failure unless you're using the *–netlet–stdio* option of Jnode. It can result in a "No Server Found: webtrix-admin" error in the launcher. The solution is to increase the maximum heap size of the JVM. Use option *–Xmx128m* on the *jnode* or *jnode.bat* command line. This will be given straight to the JVM and will increase its maximum heap size to 128MB.

B Using the launcher

The launcher allows us to run (launch) and control a Jtrix application from the command line.

B.1 Launch/management basics

Although the launch process and the management process are different, both are handled at the same console, so we use the phrases "launcher" and "console" interchangably.

To launch an application we need:

- A launch descriptor. This is a bit like a netlet descriptor but has a bit more information in it.
- A warrant for a hosting service (that's where the application will initially live, and whose resources it will consume);
- Probably a warrant for a SAS. The SAS warrant is so that anyone using the application can bind it (and hence its netlets). However, if the application wants to run its own binding then a SAS warrant isn't necessary. For example, we might be launching our own SAS service. Or we might be happy for all our warrants and netlet descriptors to contain embedded JARs, rather than containing JAR references. The significant downside of this, of course, is that they will all be several megabytes long.

The document *Programming with Jtrix: The Beatrix application framework* has more on launching applications and launch descriptors, so we won't cover them any more here.

To manage an application we need a warrant, and the application needs to support console commands.

B.2 Console commands and variables

The console allows us to execute built-in commands of its own (console commands) and applications' own commands.

B.2.1 Console variables

The console allows us to set and echo variables. Variable names are prefixed with a $ sign.

```
set greeting=Hello                  # Sets a variable
echo $greeting world                # Displays output
set v2=$greeting                    # One variable set to the value of another
number=three                        # "set" can be omitted
```

In all the examples above the variables hold strings. But variables can hold other types, too, and we force this using braces:

```
intnum={int:3}                      # Braces force a type. Here, an int
longnum={long:3}                    # This sets a long
convnum=3                           # An integer string is converted to an int
somestring={string:3}               # This forces the variable to be a string
id={x500dn:o=jtrix,ou=development}  # Set an X.500 distinguished name
w1={warrant:myfile.xml}             # Read a warrant into a variable from a file
w2={warrant:http://www.mysite.com/warrants?id=37}    # ...or from a URL
```

When we use braces to convert our entered string to a named type, that type must be supported by the `Property` class.

Notice that because a warrant is XML, a warrant variable is converted from the contents of a named file or URL—warrants are far too long to enter by hand.

B.2.2 Application commands

When connecting to an application we give it a label. Then application commands are prefixed by the application label and a "." (dot) to help distinguish them from each other and from console commands. All application commands are sent to the application to execute there. Each returns an array of values to the console. An application command can also be surrounded by backticks to capture the first element of this array:

```
connect myapp app-warrant.xml    # Connect to a running application and label it
myapp.hello                      # Execute the application's "hello" command
myapp.hello {int:123}            # Execute the command with a parameter
set var=‘myapp.goodbye‘           # Set $var to be the first return value of
                                 #     the "goodbye" command
connect app2 app2-warrant.xml    # Connect to a second application
app2.update $var                 # Execute its "update" command using the property
                                 #     obtained from the first application.
```

Each element of the returned array is also stored in the console variable bearing its index: *$0*, *$1*, *$2*, and so on. So it's *$0* that is returned by backticks.

Console variables *%0*, *%1*, *%2*, etc are set to be the extra arguments, if any, that we gave to the console when we started it. This makes it easy to pass in variables from outside the console. See also the *script* command below.

B.2.3 Application variables

As well as the console having variables, applications can have variables which we can read (but not write) from the console:

```
connect app3 other-warrant.xml    # Connect to a new application
echo $app3.somevar                # See the value of its "somevar" variable
set myvar=$app3.somevar           # Set a local variable to its value
```

In the lines above, *$app3.somevar* should be read as *$* plus *app3.somevar*. It has nothing to do with the console variable *$app3*.

B.2.4 Extended property types

As well as the built-in variable types we saw above (int, string, warrant, etc) an application can provide its own types. These are referenced a bit like application variable, but we use a *!* which appears inside braces:

```
connect app4 fourth-warrant.xml              # Connect to a running application
set n={!app4.stringarray:first,second,third}  # Set a new variable to be its
                                              # special "stringarray" type.

# Now execute its command which takes two stringarray parameters

app4.makelist $n {!app4.stringarray:fourth,fifth}
```

The interpretation of the strings *first,second,third* and *fourth,fifth* are entirely specific to that particular application and its own *stringarray* type.

B.2.5 Console command reference

Here are the key console commands. Unlike application commands console commands do not have return values:

connect <app_label> <app_warrant_file> Connect to an already-running application. The console gives it the chosen label. If we do not give a warrant file name, then we should instead use a variable which holds a warrant. The warrant is then stored in a variable with the same name as the chosen application label.

disconnect <app_label> Disconnect the console from the application. Does not stop the application.

help Get a full list of console commands.

host <warrant_file> Provide the warrant for a hosting service. When we launch a new application it will launch onto this service. If we do not give an XML warrant file name, then we should instead use a variable which holds a warrant. Puts the current hosting warrant into the standard variable $hosting.

run <app_label> <launch_descriptor_file> [<arg>=<value>]* Launch an application onto a remote node. The console will give it the chosen label for future reference. The launch descriptor file describes the netlet to be launched. Optional arguments may also be specified, and passed to the netlet. These optional arguments are application-specific. After launching, the console automatically issues a connect command. Returns a warrant for the application which can be used to connect to it again in future, and this warrant is put into the variable with the same name as the chosen label.

sas <warrant_file> Provide the warrant for a SAS. When we launch a new application it will have its JARs uploaded into this service. The warrant will also be given to the application on initialisation. If no SAS is specified before launching a new application then that application is expected to manage its own service binding; also, the netlet descriptor created by the launcher will embed the JAR files, hence making it several megabytes long.

Here are other important console commands:

debug true|false Capture all standard output from local access point netlets (if *true*) or stop this (if *false*, which is the default). Only useful if access point netlets produce standard output. Note that this is different from *dmesg*, which captures output from (remote) netlets on the (remote) hosting service.

dmesg Display latest standard I/O messages from the hosting service. In practice this means when a netlet uses `System.out.println` we can see it on the console.

dump <value> [<file>] Output a single value, such as a warrant or descriptor. If file is specified the value goes into that file, otherwise it appears on the screen. If the value is a warrant or a descriptor then the full XML is output rather than the short token that the launcher usually outputs. Useful for saving these large XML documents to a file and the passing them on to others.

echo [<value>]* Display values.

exec <app_label>.<app_command> [<arg>]* Execute the given command on the named application. The *exec* can be omitted.

list [<app_label>] List all applications, or all commands in a specific application.

quit Quits the console. Does not terminate applications.

script <filename> [<value>]* Execute commands in the given file. On entry to this file variables *%0*, *%1*, *%2*, etc will have the values specified here. Previous values of *%0*, *%1*, etc will be saved, then restored on exit.

set [<name>=[<value>]] Display all variables, set one, or unset one. Variables which carry large values, such as warrants, do not have their literal values displayed. The *set* can be omitted. If the value is omitted then the variable is unset.

sleep <seconds> Pause for a number of seconds. Useful in a script if we want to wait for an application to start up and deploy workers before continuing.

status [<prefix>] Show the status of the current hosting service (the "hosting MIB"). Can filter for only those dotted object IDs with a particular prefix.

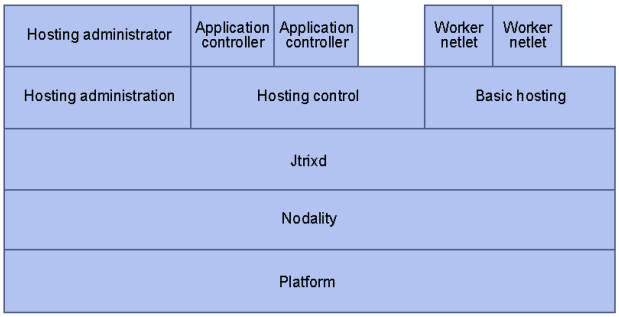

Figure 1: The Jtrix demo application stack.

C What's going on

Here we look at what's going on with these demos and how we can generalise what we see here. Figure 1 shows the application stack. The various layers are explained here, bottom up.

C.1 Platform

The platform, or base API, is the core of Jtrix, where the key classes and interfaces are defined. Anything on top of this is variable and/or optional, including the implementation of most of the elements.

C.2 Nodality

An implementation of a node. Although nodes may span several machines, Nodality is only designed to have each instance spanning a single machine. However, several Nodality instances may run simultaneously on the same machine.

C.3 Jnode

A Jtrix command line node. It adds group communication and resource facilities on top of Nodality. It also introduces the concept of "bootstrap netlets". Bootstrap netlets can be run when jnode boots,

and are granted privileged access to I/O streams. An example of this is Hospitality. It uses the group comms and resource facilities to provide a hosting service, and it outputs its admin warrant to a file (using the I/O streams) which can then be read by other bootstrap netlets.

C.4 Hosting layer (Hospitality)

This layer is just an application that sits on top of the node and provides a hosting service—i.e. it allows other applications to migrate onto this node and offers various accounting and control facilities. Jtrix.org's implementation of a hosting layer is called Hospitality and is designed to run on Jnode.

C.4.1 Hosting administration

Hosting administration allows the owner of the hosting service to create contracts for people who can then host their applications on it. The evidence for such a contract is an XML warrant generated by the hosting administration system. The contract may of course include limitations on the resources available to them on the service.

C.4.2 Hosting controller

The hosting controller is what the holder of a hosting control warrant is given rights to. It allows them to execute netlets on the hosting service, allocate resources, get their contract status, and so on.

For example, we need a hosting control warrant to launch an application onto a hosting service. Beatrix manager netlets have hosting control warrants to launch their worker netlets.

C.4.3 Basic hosting

Basic hosting is for worker netlets. Once executed they have access for this. It allows them to bind the resources allocated to them.

For example, Beatrix worker netlets use basic hosting warrants, since it gives them everything they need from a hosting service. Since they do not have authority to run other netlets they do not need a hosting control warrant.

C.5 Application layer

A layer in which applications and people make use of the hosting service.

C.5.1 Hosting administrator

The hosting administrator is a person (or application) that creates contracts for client applications and hands them warrants which gives them access to the hosting service.

C.5.2 Application administrators

For example, Beatrix manager netlets. Application administrators make use of a hosting warrant to execute worker netlets on the hosting service and allocate resources to them. Typically an application administrator is a manager netlet which co-ordinates its worker netlets.

C.5.3 Worker netlets

For example, Beatrix worker netlets. Netlets managed by the application administrator. They get their allocated resources from the basic hosting interface.

D jnode_help.txt

```
Scripts and JARs from jnode project
-----------------------------------

$Id: jnode_help.txt,v 1.3 2001/12/10 12:56:48 andyc Exp $

setjtrixip
----------

Usage: setjtrixip.bat

Calculates the ip address used on a local windows 2000 machine for the default
route. This will then be set as an environmental variable JTRIX_IP.
This code is then used by the jnode.bat to calculate the shared ip for netlets.
It creates a ipconfig.txt file for temporary processing.

jnode
------

Usage: jnode <nodenum> [-D<propertyname>=<value>]* [-jdb | -jdb2]
    [-nostdres] [-webres] [-cache-limit <size>] <extra_options>

Starts a new Nodality node with several defaults, including making
networking and disk resources available. The networking resources are
all on ports greater than 1024. Assumes all the relevant Jtrix JARs
are in the same directory as itself.

Arguments:

    -D<propertyname>=<value>
        Passed straight through to the Java interpreter which uses it
        to set a system property.
    -jdb
        Starts the Java interpreter with debugging on port 8000.
    -jdb2
        Starts the Java interpreter with debugging on port 8002.
    -nostdres
        Stops the default networking and disk resources being made available.
    -webres
        Makes extra networking resources available: ports 50-59 and 80-89
        inclusive. Not available on Windows; please look inside the script
        for details.
    -cache-limit <size>
        Limits the amount of disk space used by the cache. The size is
        specified in bytes and applies to the total size of the contents
        of the directory where the cached jars are kept. This means that
        if the cache directory is shared between several jnode's, the
        size of the cache will never grow larger than the maximum of the
        arguments given to each of them. A size of 0 means no limitatation
        to the size of the cache.
    <nodenum>
        ID number for this node.
    <extra_options>
        Passed directly to jnode.jar. This includes netlet descriptors
        and their key/value pairs.

Files:

    jnode.policy
            Java security policy used for jnode.jar.
    /var/tmp/jtrix/disk-<pid>
        Directory used for the disk space resource, where <pid> is
        the script's process ID.
    /var/tmp/jtrix/cache
        Directory used for JAR caching,
```

```
jnode.jar
----------

Usage: java -jar jnode.jar <dir> <nodeid> [-pause]
            [-disk <resid> <diskdir>]* [-static <resid> <stlabel> <stpath>]*
            [-halflife <minutes>] [-cache <cachedir>]
            [-netrange <resid> <netaddr> <min-port> <max-port>]
            [-netblock <resid> <netaddr> <min-port> <max-port> <block-size>]
            [-shared <shaddr>]* [-description <oid> <descname>]
            [-preload <jardir>|<jarfile>]*
            [-group <grport>] [-netlet-stdio] [-boot-stdio]
            [-acct] [<descriptor> [<key>=<value>]* ]* ...

Starts a new Nodality node with group communication facilities. It may
be the first node in its cluster, or may join an existing
cluster. Allows allocation of resources to nodes. Each resource is
given a resource ID integer. Different resources must be given IDs
that are unique across the cluster. Conversely, jnode can tell if two
resources are the same only if they have the same ID. Several netlet
descriptors may be specified to be started in sequence. Each of these
is called a "bootstrap netlet" and has special access to I/O streams
as specified in key/value pairs.

Arguments:

    <dir>
        Base directory in which it will run, which contains all the
        relevant Jtrix JARs.
    <nodeid>
        ID number for this node. Must be unique within this cluster.
    -pause
        Wait for keyboard input before starting up node. Allows one to
        attach a debugger before anything complicated happens.
    -disk <resid> <diskdir>
        Allocate disk space which may be made available to netlets on this
        node. It will be resource ID <resid> and map to directory <diskdir>.
    -static <resid> <stlabel> <stpath>
        Assign a read-only resource to resource ID <resid>. The resource is
        confined to volume labelled <stlabel> and is the file or directory
        labelled <stpath>.
    -halflife <miutes>
        Every <minutes> minutes there is a 50% chance of the node terminating.
        Useful for testing robustness.
    -cache <cachedir>
        Directory in which JARs etc may be cached.
    -netrange <resid> <netaddr> <min-port> <max-port>
        Allocate an IP address which may be made available to netlets on this
        node. It will be resource ID <resid> and have IP address <netaddr>.
        Netlets using this resource have access to the specified port range.
        This resource class has a single instance.
    -netblock <resid> <netaddr> <min-port> <max-port> <block-size>
        Allocate an IP address which may be made available to netlets on this
        node. It will be resource ID <resid> and have IP address <netaddr>.
        Netlets using this resource have access to <blocksize> ports from
        the specified range.  This resource class has many potential instances
    -shared <shaddr>
        All netlets are given networking facilities. This specifies the IP
        address through which the node sends all such communications. Either
        the local IP address of the machine, or 127.0.0.1.  Netlets using
        this resource are currently limited to port 1024 and greater
    -preload <jardir>|<jarfile>
        Load one or more jar files (.jar) at bootup in order to accelerate
        descriptors uploading.
        Each jar file must be preceded by '-preload'. If a directory is specified
        (instead of a single jar file) all the jar files (files with
        extension .jar) in that directory are preloaded.
    -description <oid> <descname>
```

```
        A brief description for this node. For example we may allocate
        <oid> 1 to be the hostname, 2 to be the node's physical location,
        etc. The <oid> may be a dotted integer such as "1.4.3". We may
        specify the -description several times for different <oid>s.
        The <descname> should not include spaces.
    -group <grport>
        Says which port the node should listen on for messages from
        other nodes. Must be greater than zero. All nodes in the same
        cluster should listen on the same port if they are to communicate
        successfully. This works even if two nodes are on the same machine,
        since unicast isn't used for this.
    -netlet-stdio
        Tells the node to capture standard I/O from netlets and write it
        to STDIO. Otherwise, since netlets run in their own virtual
        environments, it is discarded. Useful for troubleshooting netlets
        when we test them on a local node.
    -boot-stdio
        Tells the netlet to capture node boot information on STDIO.
    -acct
        Switch on accounting for resource usage.
    <descriptor>
        Filename of descriptor to run straight after the node boots. If there
        are several of these then they are run sequentially.
    <key>=<value>
        Define an input/output mapping for the last descriptor specified.  When
        the netlet tries to read or write through the bootstrap interface the
        filename is determined through these mappings.  Input mappings must be
        specified. Output mappings may be unspecified, in which case the output
        will be discarded.

(ends)
```

E hospitality_help.txt

```
Scripts and JARs from hospitality project
-----------------------------------------

$Id: hospitality_help.txt,v 1.1.1.1 2001/10/30 10:11:04 andyc Exp $


Introduction
------------

The JARs and scripts described here use the word "hospitality" a lot.
This hides the fact that control (ie admin) warrants for a hospitality are
actually control (ie admin) warrants for a hosting service. The
hosting service is provided by the hospitality JARs and scripts.  The
words "hospitality" and "hosting" are therefore fairly interchangable
here.


hospitality_control.jar
-------------------

Usage: java -jar hospitality_control.jar <dir> <name>

Generates a netlet descriptor so a cluster of Nodality nodes can
provide a hosting service.  The netlet is a bootstrap netlet and
creates a warrant gives netlets access to administer the hosting
service.

When the boot netlet descriptor is run in a Nodality node at boot time
it can be given arguments as in this example with the jnode script:

    jnode <jtrix_params> <name>-boot.xml [sas-in=<sas_warrant>]
           [hosting-out=<hosting_warrant>]

where <jtrix_params> are whatever parameters jtrix usually accepts.

For example, here is how to start a new node with a hosting service,
telnet and SAS.  Hospitalities admin warrant is written to a file,
clust01-admin.xml, then read by each of the next bootstrap netlets
(telnet and SAS) in sequence:

    hospitality_init clust01
    telnet_init 2000
    sas2_init niksas
    jnode 100 clust01-boot.xml hosting-out=clust01-admin.xml \
        telnet-2000.xml hosting-in=clust01-admin.xml \
        niksas-boot.xml hosting-in=clust01-admin.xml \
            hosting-out=clust01-real-admin.xml sas-out=niksas-warrant.xml \
            sas-desc=niksas-real-desc.xml

Arguments:

    <dir>
           Base directory in which warrants and descriptors are to be
           written.
    <name>
           An arbitrary string to identify hospitality. Must be
           unique within this LAN.
    <sas_warrant>
           Optional warrant to a SAS which hospitality can use.
    <hosting_warrant>
           Tells the boot netlet it should write a hosting administration
           warrant to the named file. If both sas-in and hosting-out
           are used, this written warrant will be the warrant from the
           SAS. If sas-in isn't supplied, then the warrant will be a
           boot warrant with no bind servers.
```

```
Files:

    <name>-boot.xml
           Generated boot descriptor to run a hosting controller netlet,
           which creates and manages hospitality.
    <sas_warrant>
           The input SAS warrant.
    <hosting_warrant>
           The output hosting administration warrant.


hospitality_facet.jar
-----------------

Usage: No command line usage.

JAR file of all the hosting facets. Used, for example, by
hospitality_telnet.jar.


hospitality_init
------------

Usage: hospitality_init <name> <hospitality_options>

Generates warrants for a hospitality hosting service of a given name. This
name should be unique within its LAN. Expects a public/private key
pair, but will generate them if either one is not found.

Arguments:

    <name>              Name of the cluster.
    <hospitality_options>   Passed directly to hospitality_control.jar.

Files:

    <name>.prv
                Expected/generated private key.
    <name>.pub
                Expected/generated public key.
    <name>-boot.xml
                Generated boot descriptor to run a hosting controller netlet,
                which creates and manages the hosting service.


hospitality_keygen
--------------

Usage: hospitality_keygen <name>

Runs hospitality_keygen.jar.


hospitality_keygen.jar
------------------

Usage: java -jar hospitality_keygen.jar <name>

Generates a public/private key pair. This can be used for any application,
not just hospitality

Files:

    <name>.prv  The private key of the pair.
    <name>.pub  The public key of the pair.
```

```
hospitality_telnet.jar
------------------

Usage: java -jar hospitality_telnet.jar <dir> <port> [<outfile>]

Used by script telnet_init. See that for more details.

Arguments:

    <dir>
           Base directory in which warrants and descriptors are to be
           written.
    <port>
           Port which the telnet server should listen on.
    <outfile>
           Filename of XML netlet descriptor generated.

Files:

    <outfile>
           Descriptor generated.


telnet_init
-----------

Usage: telnet_init <port>

Generates a bootstrap descriptor for a telnet server which provides
telnet access to the hosting service. This is for use by the hosting
administrator only.

This telnet server will expect an input parameter ("hosting-in") on
boot which is the hosting admin warrant to expose using telnet.  The
telnet server will run on port <port>.

Here is an example of how to create a descriptor for a telnet server on
port 2000, then start a new node (with ID 100) in a new cluster "fred"
with this server. The hosting admin warrant will be in a file called
"fred-boot-admin.xml":

    hospitality_init fred
    telnet_init 2000
    jtrixd 100 telnet-2000.xml hosting-in=fred-admin.xml

Arguments:
    <port>
        Port which the telnet server should listen on.

Files:
        telnet-<port>.xml
            Generated descriptor.

(ends)
```

F jtrixmaker_help.txt

G launcher_help.txt

H sas2_help.txt

```
Scripts and JARs from sas2 project
----------------------------------

$Id: sas2_help.txt,v 1.6 2001/10/19 11:44:12 nik Exp $

sas2.jar
--------

Usage:   java -jar sas2.jar <dir> <app_desc_in> <boot_desc_out>

Creates descriptor to start a SAS service.

Arguments:

    <dir>
        Base directory in which it will run, containing all the relevant
        Jtrix JARs.
    <app_desc_in>
        Required application descriptor of the SAS.
    <boot_desc_out>
        Generated netlet descriptor of SAS, to be used on booting
        a hosting service to give it a SAS. The script sas2_init creates
        this as <name>-boot.xml. See the example there for more details.


sas2_init
---------

Usage: sas2_init <name>

Creates descriptor to start a SAS service, using sas2.jar. Expects
a public/private key pair in <name>.pub and <name>.prv, but will generate
them if not found.

The resulting netlet descriptor, <name>-boot.xml, expects parameters
as in the following example (see below for argument details):

    jtrixd <node_id> <usual_hospitality_parameters> \
        <name>-boot.xml hosting-in=<h_warr> [hosting-out=<h_out>] \
            [sas-out=<s_warr>] [sas-desc=<s_desc>]

Therefore, here is an example of how to start a new node in an
existing hosting service so that it starts with SAS:

    hospitality_init fred
    sas2_init jim
    jtrixd 100 fred-boot.xml hosting-out=fred-admin.xml \
        jim-boot.xml hosting-in=fred-boot-admin.xml \
            hosting-out=fred-real-admin.xml sas-out=sas-warrant.xml

Arguments:

    <name>
        String name of the SAS service. Should not contain spaces.
        Should not be the same as the hosting service name.
    <h_warr>
        Required hosting controller (admin) warrant.
    <h_out>
        Generated hosting admin warrant, for a hosting service which
        uses this SAS.
    <s_warr>
        Generated SAS admin warrant.
    <s_desc>
        Generated SAS netlet descriptor, which uses itself as a SAS.
```

```
Files:

    <name>.prv
        Expected/generated private key for the SAS.
    <name>.pub
        Expected/generated public key for the SAS.
    <name>.xml
        Generated launch descriptor to launch SAS via the launcher
        console.
    <name>-boot.xml
        Generated netlet descriptor to start SAS when a node starts.

(ends)
```

I dns_help.txt

J http_help.txt

K webtrix_help.txt