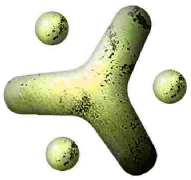

Jtrix: How to write netlets

March 2002, for release 0.14.4 and later

$Id: writing-netlets.lyx,v 1.26 2002/01/23 11:45:09 nik Exp $

Nik Silver, feedback@jtrix.org

Jtrix Ltd, 57–59 Neal Street, Covent Garden, London WC2H 9PJ, UK

Contents

Chapter 1

Introduction

This document is a complete introduction to Jtrix and how to write netlets for it. It covers a very basic "hello world" application in three different forms, use of system resources, netlet innards and technique. It is a companion to *Start running Jtrix*, available from `http://www.jtrix.org` which describes how to run various Jtrix applications.

1.1 How to read this document

Here are the chapters, and how you might like to read or skip them:

- Chapter 1: this chapter includes how to write a simple netlet, without much explanation.
- Chapter 2: high level concepts, and highly recommended as an introduction.
- Chapter 3: a more program-level view of Jtrix. Highly recommended for the Jtrix programmer.
- Chapter 4: the main packages, classes and interfaces. Worth understanding broadly at first and coming back to as needed.
- Chapter 5: the first netlet again, but this time discussed in much more detail. This is where you get to test a lot of your know-how.
- Chapter 6: how to write a Jtrix service. Complements the last chapter which dealt with the client. At this point you should have a full, if introductory, view of Jtrix.

- Chapters 7 and 8: more simple netlets, this time showing off further aspects of nodes and netlets.
- Chapter 9: Debugging and troubleshooting tips.
- Chapters 10 and 11: more esoteric aspects of Jtrix, but even a quick read of this will give you a very good understanding of its deeper power.

1.2 A very small netlet

Note: *At the current time Jtrix Ltd is not running a public "hello world" service so this example cannot be run. However, the example has been left in for demonstration. Please do read it. Later we will see how to write our own service, so the lack of a public "hello world" service will not matter.*

Despite the fact that we've not introduced any terms yet (don't worry, we will), we present here a very small Jtrix program, or "netlet", with lots of comments. The point of this document is to explain the program, and whole lot more. Meanwhile, here's a brief summary.

The netlet takes a warrant, which is a right to use a service, and binds (i.e. connects) to it. In this case the service is a hello world service, so it gets the service's message interface (called a "facet") and then gets the message. Also, there's a lot of things it doesn't do, so several other methods are included to say as much—they just return redundant values or throw exceptions.

There are few imports, because this isn't a big application:

```
package org.jtrix.project.helloworld;

import org.jtrix.base.*;
import org.jtrix.project.libjtrix.netlet.NullService;

/** Netlet which accesses a hello world service using a warrant
 * read from an incoming parameter.
 */
public class Hello1Client implements INetlet
{
```

All the work is done when the netlet initialises, like this:

```
    public byte[] initialise(INode node, Object param, byte[] unsigned)
        throws InitialiseException
    {
        try
        {
            // Get the warrant, use it bind the service, and then get the facet
            Warrant warrant = (Warrant)param;
```

```
            IService service = node.bindService(warrant, new NullService());
            String hf_name = IHelloFacet.class.getName();
            IHelloFacet facet = (IHelloFacet) service.bindFacet(hf_name);

            // Use the facet
            System.out.println(facet.getMessage());
            return null;
        }
        catch (Throwable e)
        {
            System.out.println("Sorry, no message available");
            e.printStackTrace();
            throw new InitialiseException(e.toString());
        }
    }
```

And we need to implement a few more methods, but we do so only minimally:

```
    public void terminate(long date, INetlet.IShutdownProgress progress)
    {
        // Nothing to clean up when we terminate
    }

    public IService bindService(Warrant warrant, IService consumer)
        throws ServiceBindException
    {
        // This netlet offers no services to other netlets using warrants
        throw new ServiceBindException();
    }

    public String[] getFacets()
    {
        // This netlet offers no facets to the node
        return new String[0];
    }

    public IRemote bindFacet(String facet) throws FacetBindException
    {
        // The node shouldn't try to bind facets from here
        throw new FacetBindException();
    }
} // Hello1Client
```

The interface `IHelloFacet`, mentioned above, just looks like this. This is how our netlet talks to the hello world service.

```
package org.jtrix.project.helloworld;

import org.jtrix.base.*;

public interface IHelloFacet extends IRemote
{
    public String getMessage();
}
```

1.3 Compiling our code

To compile these two classes we need to make sure *jtrix.jar* and *libjtrix.jar* are on our classpath. Then we bundle both *.class* files into a

JAR called *hello1.jar*. That's it; we don't even need to include a manifest.

1.4 Running the netlet

Now a netlet needs to run in a Jtrix runtime environment, called a "node". Jtrix.org has implemented a command line node called "Jnode". Don't run this yet, we aren't quite ready for it, but here's what this would look like when we run it with Jnode:

```
% jnode 201 -netlet-stdio hello1-client.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting hello1-client
netlet:201.0.1: Hello, world, and thank you for using our service
Bootstrap complete
^C

%
```

We need to hit Control-C to terminate Jnode.

The hello world service is a real external service, but we don't need to write any network code. And access to it was achieved by mobile code downloaded and running safely within the Jnode node, but that was all handled for us. Clearly the message the netlet got from the service is not very exciting, but it's a start.

But what is that XML file? And where did it come from? Well, a netlet is described by means of a "netlet descriptor", which is the XML file. This describes the netlet to the node, so the can run it. Where did it come from? That's next. . .

1.5 Creating a netlet descriptor

Here is how we generate that XML netlet descriptor. A netlet descriptor simply says what JARs a netlet needs to run, where its start point is and any parameters it needs. The one parameter it does need is an XML warrant. We can download this from `http://www.jtrix.org/warrant/hello-warrant.xml` and we'll save it into a file called *hello-warrant.xml*.

To create the netlet descriptor we need to say what type of XML file we're generating (a netlet descriptor), what the main class is, what JARs are needed and what directories hold them, the parameter (the warrant) for start-up, and what we the resulting XML file to be called:

```
% ls
hello-warrant.xml  hello1.jar
% jtrixmaker -type netlet -outfile hello1-client.xml \
     -jardirs /usr/lib/jtrix . -jars libjtrix.jar hello1.jar \
     -classname org.jtrix.project.helloworld.Hello1Client \
     -param {warrant:hello-warrant.xml}
% ls
hello-warrant.xml  hello1-client.xml  hello1.jar
%
```

You might need to change the Jtrix binaries directory for your system. In the end we have the netlet descriptor *hello1-client.xml*.

Now we can run the netlet with Jnode as shown above.

As a final comment, some people have found problems with this example if they are working from behind a fussy firewall, and if so then all such examples will exhibit the same problems. The current solutions are (1) to run such examples from a location without such a firewall, or (2) jump to the examples in Chapters 6, 7 and 11 which are local only, and do not require external connections. Also, look on `http://www.jtrix.org` which will have some discussion on this issue.

1.6 Summary

We've seen a lot, with only the briefest of explanations. We've seen what a Jtrix program (a "netlet") looks like, and how we can describe it as XML (its "netlet descriptor"). And we've seen how we can run it, and seen it attach to an external service and use that service.

Of course, this is not meant to be crystal clear right now. And that, and more, is what the rest of this document is all about. . .

Chapter 2

High level concepts

A primer for Jtrix concepts, from the ground up. These don't directly relate to code, but explain the principles behind it all.

2.1 Services

Any on-line activity performed for others and which is generally available is considered a Jtrix *service*. By "generally available" we mean that the business in general is the service, not a single transaction or a series of transactions taking place under a single agreement.

A service includes activities such as Web-based mail, credit card authorisation or news headlines. But a service also includes access to resources such as disk space, memory, CPU time, bandwidth and IP addresses.

2.2 Nodes and hosting

Any Jtrix application runs in a Jtrix environment which manages various services. We can imagine that this environment is based in a single server or PC. But this need not be the case—it could physically exist over several machines working together.

For this reason we do not refer to a Jtrix environment as a server or a PC. We call it a *node*. A node is one or more machines working together to provide a single environment for applications. It may be run by an individual, a department or an ISP. Regardless of whether a node is one

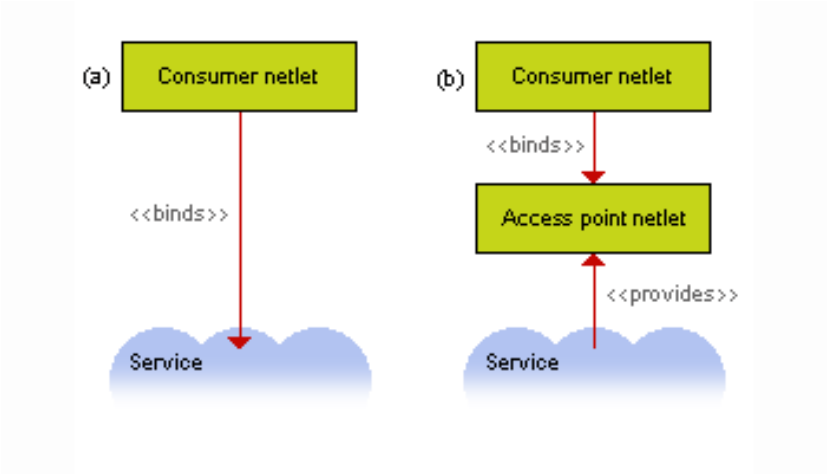

Figure 2.1: (a) A netlet uses a service. But the reality is that (b) the service is accessed via another netlet.

machine or one thousand the application does not need to worry about it.

Nodes can offer services. One key service they can offer is *hosting*. By making use of a hosting service a Jtrix application can move from one node to another.

2.3 Netlets

A Jtrix application is made up of one or more program components. There will often be several core components. Each time one buys into a service the request is accessed via another component which attaches itself to the original one. Each such component is called a *netlet*. A netlet is a program or program-fragment. See Figure 2.1.

Any one netlet may use any number of other netlets to perform its task, or it may be stand-alone. Thus services are accessed through netlets. But a service is more than just a lot of netlets; the netlets are only the customer-facing part of what is probably a much larger operation.

At the code level a netlet is just an object that implements an interface (called `INetlet`) and therefore has a few standard methods.

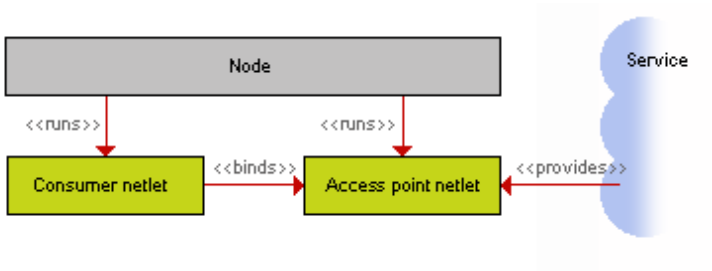

Figure 2.2: A client netlet binds to a service through another netlet. The netlet-to-netlet binding creates a service session. Both netlets need to sit on the same node. The service is fufilled through the netlet, and is more than just the netlet itself.

2.4 Binding and service sessions

When one thing attaches to another it is called *binding*.

For a netlet to run it must be bound to a node. More precisely, the node must execute it by binding the netlet to it; the netlet is a useless lump of data until the netlet binds it. See the vertical arrows in Figure 2.2.

For one netlet to use a service provided by another it must bind to the service. The relationship created by the binding is called a *service session*. Note that when one netlet uses another they must both be on the same node. See the horizontal "binds" link in Figure 2.2. If a netlet downloads to provide services then it is also called an *access point netlet*. An access point netlet may act as a gateway to facilities beyond the node, or it may provide some or all those facilities itself.

So the summary so far: A node is a runtime Jtrix environment which can span several machines. Netlets are programs or program fragments which bind to each others' services. They must exist on the same node in order to bind.

2.5 Nodality

A node must conform to certain communication standards. Our node implementation is called *Nodality*. 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.

Nodality is actually a very minimal, do-nothing node—doesn't even have any kind of user interface. Instead, it's designed to be embedded in other applications, as we will see next.

2.6 External netlets, launcher, Jnode and bootstrap netlets

We've just said that a node does not have any user interface, and it does not have any direct access to the outside world. This would make it rather difficult to control, and even boot.

Therefore we have the concept of an *external netlet*. This is any ordinary application which happens to have a node embedded in it. It can boot the node, and the node sees it as another netlet, hence the name. This external netlet is the administration netlet and has special access to the node and its resources.

External netlets are therefore how a Jtrix node can be part of another application. In fact, a node cannot run without an external netlet.

One example of an external netlet is *launcher* which is jtrix.org's shell-like console tool to access and manage several Jtrix applications from a single remote location. It has a node embedded in it so it can run small netlets which send the user's commands back to the main Jtrix application to process.

Another example is *Jnode*, the node with a command line start-up. This enables a single instance of Nodality to act as part of a group and helps resource management.

When we start Jnode we can specify a number of netlets it should initialise. These are called *bootstrap netlets*. We can grant them I/O access to specific files, which security would otherwise prevent. Let's not confuse bootstrap netlets with external netlets—Jnode is the external netlet, because it's the application which embeds the node, and the external netlet controls the node. The bootstrap netlets are the ones given on the Jnode command line, which it controls.

2.7 Netlet descriptors

A *netlet descriptor* is an XML file which tells a node enough for it to load and bind a netlet. Among other things it must tell the node where to get the code from, so it will contain either URLs for the netlet code, or the code itself, or a combination. See Figure 2.3.

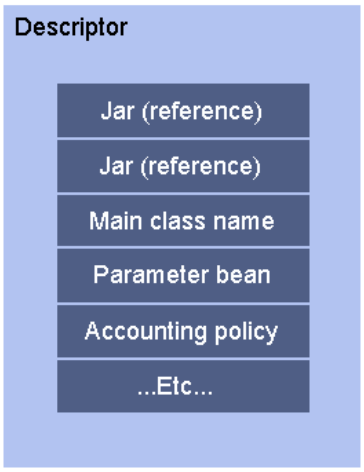

Figure 2.3: A descriptor contains enough information to load and run a netlet. That includes the codebase (or references to it) and an initialisation object among other things.

This code, or its URL, allows the node to load the netlet. To bind it, the netlet's initialisation method will be called, but this may require some parameters. So a descriptor also contains a parameter object which it passes to the initialisation method.

Thus a descriptor contains code (and/or URLs for the code) plus an initialisation parameter, and this is enough to load and bind a netlet.

A descriptor is also a key part of a hosting service, when a netlet wants to start up on a new node. By being provided with the netlet's descriptor the new node can create a copy of the netlet and initialise it. On initialisation the netlet can do whatever it needs to run.

2.8 Warrants

A warrant is a right to use a service.

Two of our hello world examples have a client netlet and a server netlet; the client gets and displays the message from the server netlet. The clients have warrants which allow them to use a hello world service.

Another way to think of a warrant is that it is evidence of a contract. If a contract has been arranged (on- or off-line) then a warrant will result so that the service can be used and the contract fulfilled.

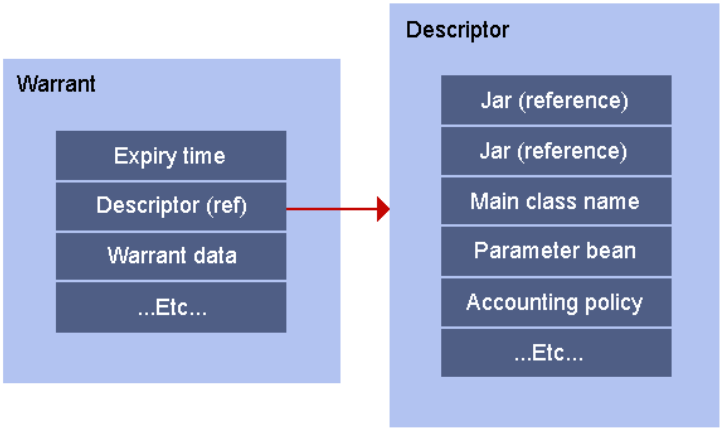

Figure 2.4: A warrant is a right to use a service. Among other things it must contain a descriptor for the relevant netlet, or a reference to one.

When a client netlet wants to use a service it presents its warrant to the node which then finds and binds the server netlet. On binding, a service session is created. In fact, service sessions can only be created though warrants.

Of course, for a node to use a warrant to find a service the warrant needs to contain either a descriptor or a URL reference of where to find that descriptor—Figure 2.4. So to make use of a service, a netlet needs only a warrant. This contains everything a node needs to find and bind a service, however indirectly.

To recap: a descriptor contains netlet code, or a URL for netlet code, plus an initialisation object. A warrant contains a descriptor or a URL for a descriptor.

A warrant is not just a general right to use a service—it is very likely to be limited for the terms of the particular contract for which it was issued. Two netlets may have warrants to use a service, but one may be time-limited, and another may be resource limited. That's up to whoever arranged the contract. In Figure 2.5 different warrants for the same service allow their holders access only to their area of that service.

The agreements that warrants are based on are key to Jtrix, because nothing can execute without explicit rights to do so. Moreover, all resources consumed (bandwidth to load the netlet, CPU time and memory

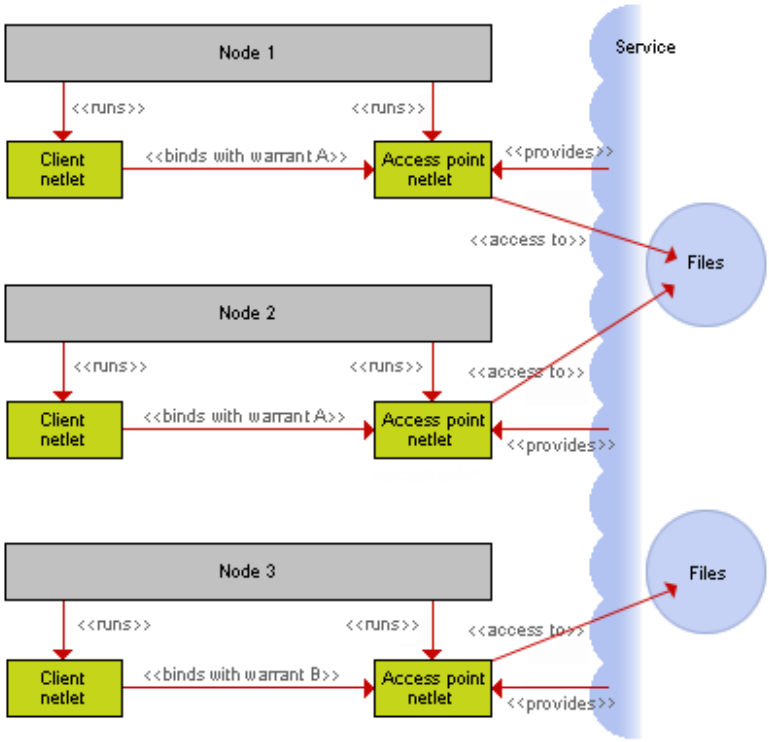

Figure 2.5: Warrants A and B are both for use of a file storage service. But warrant A limits its holder(s) to a specific group of files, while warrant B is limited to a different group. Thus each warrant-holder has access only to their files.

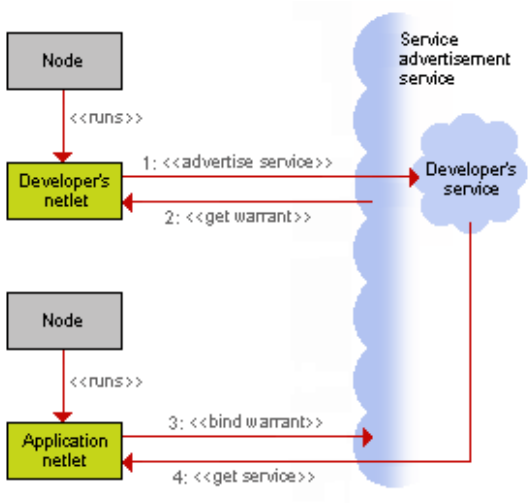

Figure 2.6: Using the service advertisement service (SAS). (1) A developer advertises their service to the SAS. (2) In return they get a warrant which they give to their application netlets. (3) and (4) An application netlet uses the warrant to bind to the developer's service.

to run it, disk space to store it, etc) must be accounted for and, almost certainly, paid for.

2.9 The service advertisement service (SAS)

When a developer creates a service they need to make its access point netlets "generally available". If they write the service on their home PC or behind a firewall then its lifetime and availability is limited.

A *service advertisement service* is just a download point for netlet descriptors and JARs (so "advertise" is admittedly misleading). So once our developer uploads their netlet descriptors and JARs they no longer need to stay connected to the network; their netlets are still available.

After uploading, the developer gets a warrant which points to the upload locations (now download locations) of the descriptors and JARs. They can give this to client netlets which want to use the service. These clients use the warrant to bind to the developer's service. The fact that it is handled by the SAS is irrelevant to them. The owner of the SAS is responsible for keeping it generally available. See Figure 2.6. Notice

that we do not define how the developer gives the warrant to the client netlets—that's down to the contract between them.

Depending on the sophistication of the SAS it may allow static descriptors to be advertised (and delivered) or perhaps something more interesting, where the descriptor delivered varies based on particular circumstances. See also Section 3.6.

2.10 Summary

Let's try to use all our jargon in one paragraph. . .

Jtrix nodes span several machines, and offer services to netlets including hosting. When a node hosts a netlet it needs to load and bind it. This is achieved by virtue of the netlet's descriptor which includes its code (or a reference to it) and any initialisation object it needs. Services are also offered through netlets for other (client) netlets, but a client netlet cannot bind to a service without a warrant to do so. Services may be delivered through service advertisement services, after having been placed there by their creators. An external netlet is one with a special link to the outside world and the node; it is usually the system administrator's interface onto the node, and the one which originally booted it. Jnode is an external netlet for Nodality, which adds communication and resource management.

If you can follow that then we are ready to go.

Chapter 3

Facets and interfaces

Our programming concepts here boil down to one thing: interfaces. And we do mean just ordinary common-or-garden Java^{TM} interfaces that occur in every Java application.[1] In Jtrix we have various kinds of interface, and we explain them here.

3.1 Facets, binding and services

As we know, netlets provide services. In programming terms, a consumer is given a service interface by a netlet. See Figure 3.1(c) which is a refinement of Figure 2.1.

A *facet* is an interface to use a feature of a service.

A service can be quite complex and may need serveral facets. For example, a service may offer two facets: one facet for general use and one for usage accounting. A storage service may offer one facet for file read/writes and one for transactions. Our hello world service offers one facet, through which the message is obtained.

Just like services, a facet needs to be *bound*. A netlet cannot get a hold of a facet without binding to it, but once bound it can use it just like any other object.

So the general procedure for a netlet to grasp a facet is: (1) bind to a service; (2) get its list of facets; (3) bind to one of them. Step (1) is achieved just by presenting a warrant to a node, and if the netlet already knows the name of facet it wants then it can skip step (2), so the the procedure is really (1) bind to service; (2) bind to facet.

[1] Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

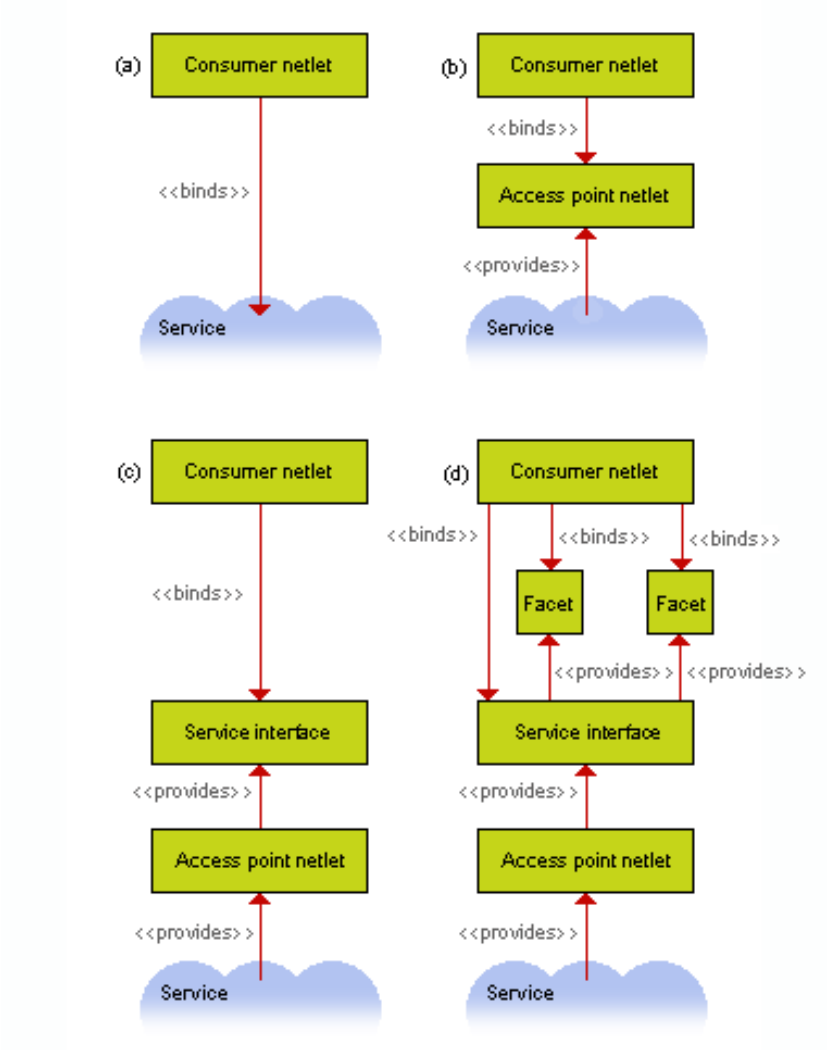

Figure 3.1: A refinement of Figure 2.1. As we've seen, (a) a consumer netlet binds to a "generally available" service which (b) is really accessed via another netlet. But really (c) the consumer gets an interface representing the service. Following this (d) the consumer netlet binds to facets which allow use of the service's features.

Figure 3.1(d) is a refinement of Figure 3.1(c). It shows how netlets provide services which provide facets.

3.2 Sessions and servers

Any kind of binding creates a *session*. As mentioned in Section 2.4 binding to a service creates a *service session* or *service connection*. Binding to a facet also creates a session, but of course we do not call it a service session—service sessions are only created with warrants.

Because netlets and services are different—netlets fulfill services—we often refer to netlets which deliver services as *servers*. For example, there may be many netlets which offer access to one service; they are alternative servers to the same service. The distinction between server and service is a fine one, but it is accurate and helpful.

3.3 Netlet facets

Facets do not only come from services. Netlets can offer facets directly, too. These are called *netlet facets*.

Only nodes (and the privileged external netlet) can access netlet facets. A netlet facet is an interface onto some part of the netlet which only the node and its external netlet can access. See Figure 3.2.

An example of a netlet facet is the one called `ITextStatusFacet`. This is a facet which Nodality uses to get status messages from the netlet. When a netlet is bound to a Nodality node it checks to see if `ITextStatusFacet` is there, and if so binds to it. Then whenever the netlet sets the status string it can be picked up by Nodality which can then display it. `ITextStatusFacet` is a convenient way for the netlet to communicate with the outside world and is a good example of a netlet facet.

3.4 Node facets

In addition to service facets and netlet facets, nodes can offer facets, too. These are called *node facets*.

Node facets are available to netlets running in that node, and provide a way of nodes offering facilities to netlets.

For example, one facet offered by Nodality is its “execution” facet. Through this a netlet can terminate other netlets, groups, and so on. However,

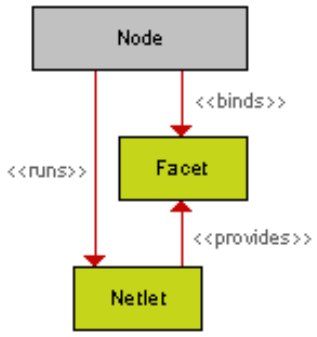

Figure 3.2: A netlet offers a netlet facet to the node, to allow some more specialised form of communication. A session is created.

this is generally available only to the external netlet since that is the privileged system administrator's connection.

Also Nodality offers an administration facet through which any netlet may choose to add to the node facets already available. This is what one of our hello world examples uses: an initial netlet provides the node with its hello facet. A client netlet can then pick it up from the node—see Figure 3.3.

It is worth mentioning that we don't require execution and administration facets to exist on a node. It's something particular to Nodality. Nor do we insist that netlets can add to the node facets currently available. Again, this is a feature of Nodality. But it's a very useful feature. For example, Nodality could offer a hello world service as standard, and have this service hardcoded into it, but it doesn't seem very future-proof. More to the point, it could offer a banking service as standard, but banking standards and affliliations are bound to change and may not always be needed. Nodality's solution is to allow these services to be supplied by netlets which in turn provide them to the node. When needs change it is just a matter of changing the netlet, not rewriting Nodality. This elegant "plug-in" architecture gives us a node which is small, expandable and futureproof.

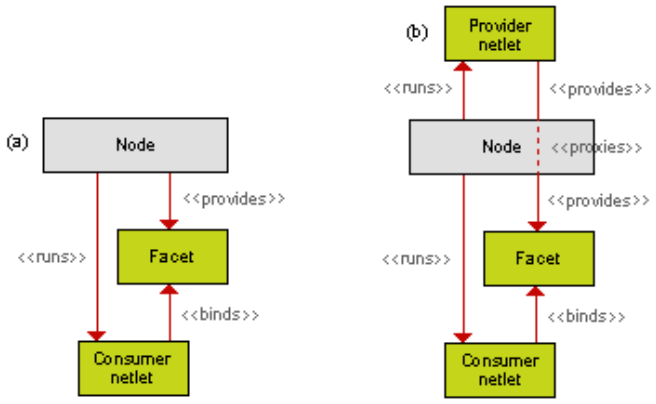

Figure 3.3: Consumer netlets bind to node facets. In (a) the facet is provided directly by the node. In (b) the node gets the facet from a provider netlet, although this is invisible to the consumer netlet.

3.5 Mediation and security

Service sessions are not direct connections between netlets; that would not be secure enough. Instead the node *mediates* sessions, ensuring local security requirements are enforced. What a netlet uses is actually a proxy. The proxy runs in the netlet's own class space; the real work is done by the netlet/service/node running in its own class space. The mediation is performed by the node's *mediator*.[2]

In fact, the node mediates all links between a netlet and the rest of the world, including networking and file system use. Figure 3.4 is a refinement of Figure 3.1(d). It shows where the mediation boundary is between class spaces.

For auditing and security, and because mediation is necessary, a facet or service cannot live without its provider netlet. In Figure 3.4 if the access point netlet dies then the service connection is terminated and its facets become useless to the consumer netlet. Similarly in Figure 3.3 if the provider netlet dies then all its facets go with it.

[2]A bug in Sun's JDK 1.3.0 prevents chars from being proxied. This has been corrected in JDK 1.3.1. See `http://developer.java.sun.com/developer/bugParade/bugs/4346224.html`

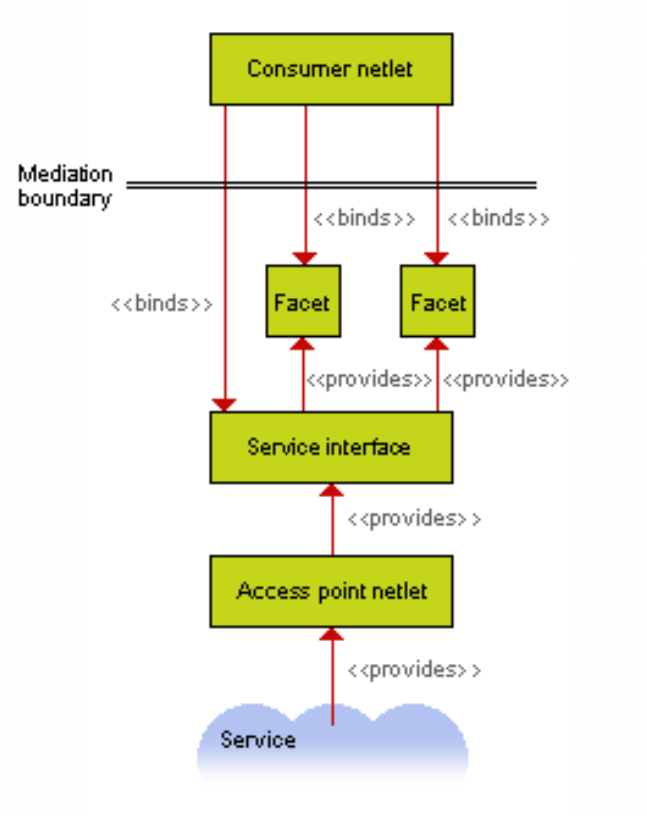

Figure 3.4: A refinement of Figure 3.1(d). All binding is mediated by the node.

3.6 Binding protocols and unsigned data

As we know, use of a warrant creates a local netlet to access that service. A descriptor tells the node how to create and run this netlet.

The descriptor may be embedded in the warrant, but usually it will contain a URL reference to it; this allows the descriptor to be created on the fly if necessary. For example, it may contain parameters specific to current circumstances, or the codebase may even be different.

The node fetches the descriptor across the network using a *binding protocol*. If the URL is just an HTTP reference then of course the binding protocol is a simple HTTP fetch. But the URL may be anything, and may be a special protocol that allows negotiation—for example exchanging private keys, establishing resource availability, and so on.

Furthermore, when the binding server delivers the descriptor it may deliver additional data alongside it. Data embedded in the descriptor may be digitally signed, but the additional data is not: it is *unsigned data*. When the netlet is created and run it is given both kinds separately, and it can treat them each appropriately.

Unsigned data allows the server to send dynamic data without the overhead of compiling it into a descriptor. Also, descriptors are language-dependent (currently only Java), but unsigned data is language independent. This also explains why it is presented to netlets as a byte array; anything else is language-specific.

3.7 Conventions, INetlet, INode, IService and IRemote

It is a Jtrix convention that interface names always start with the letter `I`, while facets tend to end with `Facet`. Hence our hello world service facet is called `IHelloFacet`, while the node's administration facet is called `INodeAdminFacet`.

Now we can talk about some of the key interfaces in use (Figure 3.5). Note what follow are just interfaces, not facets—facets are extracted from these.

`INode` is used by netlets to talk to their hosting node. Primarily a netlet uses this to bind services using a warrant. But it is also used if the netlet wants to bind to node facets.

`INetlet` is the counterpart of `INode`. It lets the node talk to the netlet, and without implementing this interface a class is not a netlet.

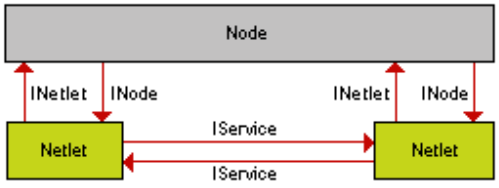

Figure 3.5: Netlets present INetlet interfaces to nodes. Nodes present INode interfaces to netlets. Netlets present IService interfaces to other netlets.

Methods it must implement are `initialise()` for when the node initially binds it; `terminate()`, a notification that the node is terminating it; `bindService()` for when another netlet wants to make use of it; `getFacets()` for the node to find out what netlet facets it provides; `bindFacet()` for the node to bind to one of these netlet facets.

`IService` allows netlets to communicate with each other. When a client netlet binds to a service each gets the other's `IService` interface in return. Through this interface either one can terminate the relationship or bind to the other's facets. Thus the `IService` interface includes the methods `terminate()`, `getFacets()` (to allow another netlet to find out what service facets are available) and `bindFacet()` (to allow another netlet to bind to one of them). `IService` is a good name for this interface because strictly speaking netlets don't bind with each other—they bind to their services. However, it's important to realise that `IService` represents a single connection to a service, it does not represent the entire service itself.

`IRemote` is the top level super-interface of all service interfaces. When a service is bound the `bindService()` method returns an `IRemote` object, and the client then casts this into whatever type is needed.

3.8 Interface summary

- Nodes bind to netlets
- Nodes bind to netlet facets
- Netlets bind to node facets

- Netlets can add to Nodality's collection of node facets.
- Nodes talk to netlets through the `INetlet` interface
- Netlets talk to nodes through the `INode` interface
- Netlets fulfill services
- Services offer facets
- Netlets bind to services
- Netlets bind to service facets
- Netlets talk to each others' services through the `IService` interface

Chapter 4

Packages, JARs, classes and methods

Only key details are presented here. Please see the javadoc for more.

4.1 Packages

Table 4.1 shows key packages and what they're for. The most important thing to understand that only `org.jtrix.base` is considered to contain the core Jtrix classes and interfaces. Having this means you have Jtrix. Going on from here a particular Jtrix implementation may implement a different grouping system, library, etc. The ones we have presented here are only the systems implemented by the Jtrix.org team.

4.2 JARs

Table 4.2 shows some of the JARs provided by Jtrix.org. There are a couple of things of particular interest.

First, *jtrix.jar* is guaranteed to be available on any node on which a netlet runs. That means when a netlet's descriptor lists what JARs it needs it can omit mentioning *jtrix.jar*—it will already be there for it.

Second, JARs such as *hospitality_facet.jar* and *nodality_facet.jar* often crop up, and they contain the just the facets that a service offers. They are very useful to compile and run against because they are small and leave the actual implementation to the services themselves.

Package	**Description**
`org.jtrix.base`	Jtrix definition. Everything else is optional and is designed to be substituted if needed.
`org.jtrix.facets1`	Key Jtrix facets and small supporting classes. This all goes into *facets1.jar* and will remain constant. Facets2, 3, etc will appear in future.
`org.jtrix.facets1.netlet`	Netlet facets: status facets and bootstrap facets
`org.jtrix.facets1.node`	Node facets: resource trampolining and auditing.
`org.jtrix.facets1.util`	Utility APIs: I/O, properties, networking etc.
`org.jtrix.facets1.service`	General service user APIs which all services of a given type should support. Includes SAS and hosting.
`org.jtrix.project.libjtrix`	A library of generally useful classes and methods.

Table 4.1: Key Jtrix packages.

JAR	**Description**
jtrix.jar	All of `org.jtrix.base`.
facets1.jar	All of `org.jtrix.facets1`.
libjtrix.jar	All of `org.jtrix.project.libjtrix`.
beatrix.jar	The Beatrix application framework. All of `org.jtrix.project.beatrix`.
parser.jar	Various classes from `org.xml`, `org.w3c` and `com.sun` needed for XML reading and writing.
jaxp.jar	Sun's Java API for XML parsing. Needed for reading XML.
hospitality_facet.jar	Facets offered by the Jtrix.org hosting service, "hospitality". We can compile against this JAR if we are exploiting that service.
nodality_facet.jar	Facets offered by Nodality. We can compile against this JAR if we are exploiting that service.

Table 4.2: Key Jtrix JARs.

4.3 Classes and interfaces

These classes and interfaces are presented in alphabetical order. Therefore this section is to be used as needed, rather than to be read from start to finish.

4.3.1 org.jtrix.base.FacetHandle

A `FacetHandle` wrapper should be created to return a facet in response to a request for one. E.g. in response a `bindFacet()` call.

```
public class FacetHandle implements IRemote
{
    /** Create a facet handle.
     * @param impl  The implementation for the facet.
     * @param facet  The class name of the facet.
     */
    public FacetHandle(IRemote impl,String facet)

    // Other methods omitted
}
```

The `FacetHandle` is quite trivial but very important. If ever we need to return a facet we should wrap it in a `FacetHandle`. It helps the node understand what we're passing back, so it can proxy it properly through its mediator.

So this is a facet implementation plus a hint to the node as to how it should be proxied across the mediator to the requester. If we didn't return a `FacetHandle` and instead passed the actual implementation then the node's mediator wouldn't have enough information to create the proxy which the requester receives.

For example of its use, see the method `FacetProvider.bindNodeFacet()` in Section 7.3.

4.3.2 org.jtrix.base.IFacetCollection

This is simply a representation of any single collection of facets such as `INetlet` (a netlet may offer facets directly to a node) or `IService`.

```
public interface IFacetCollection extends IFacetProvider
{
    /** Fetch names of available facets.
     * @return  The names of all the facets in this collection.
     */
    public String[] getFacets();
}
```

4.3.3 org.jtrix.base.IFacetProvider

An interface to anything which provides a facet, such as a node (which offers facets to netlets) or a service (which also offers facets to other netlets).

```
public interface IFacetProvider extends IRemote
{
    /** Bind to a facet by name
     * @param facet  The name of the facet. This should be the class name of
     *     the facet and the interface should be in the consumer's class space.
     * @throws FacetBindException  If an error occured while binding to the
     *     facet.
     * @return  The remote interface. This can then be cast into the interface
     *     of the required facet.
     */
    public IRemote bindFacet(String facet) throws FacetBindException;
}
```

4.3.4 org.jtrix.base.INetlet

This is the interface which allows the node to communicate with a netlet. Notice that the `initialise()` method can return any arbitrary data. This is for when one netlet runs another; then the executing netlet can get back anything in return. But we don't need this if we're running a netlet from Jnode, because it's just interested in a successful start, and that means not throwing an exception.

```
public interface INetlet extends IFacetCollection
{
    /** Allows the node to initialise the netlet. Called at most once.
     *
     * @param node  Interface to the netlet's view of the node.
     * @param parameter_bean   Bean object for parameters from the descriptor.
     *     Since this comes from the descriptor, if the descriptor is signed
     *     then we can assume this data is equally trusted.
     * @param unsigned_arg  Arbituary data that arrived with the descriptor,
     *     but external to it. Hence it is not signed (unlike the
     *     <tt>parameter_bean</tt>) and it is therefore untrusted.
     * @throws InitialiseException  In case of failure an exception is thrown;
     *     the netlet can then expect to be terminated.
     * @return Some arbitrary data; it is determined by the implementation and
     *     the context; it must be properly intepreted by whoever executed
     *     this netlet. Null is a valid, non-error return.
     */
    public byte[] initialise(INode node,
                             Object parameter_bean,
                             byte[] unsigned_arg) throws InitialiseException;

    /** Allows the node to request termination of netlet.  Called before
     * netlet is actually shutdown regardless of initialisation state. May be
     * aborted without notice if takes too long. No threads should be left
     * when this function returns, otherwise they will be shut down, too.
     *
     * @param time  The time when the node will force a terminate even if the
     *     netlet has not yet terminated. This is seconds since 1 Jan 1970
     *     GMT.  When negative there is no fixed time, but the node may then
     *     terminate at any time. If this number is 0 then the node is
```

```
     *     informing the netlet that connections may be terminated
     *     asynchronously.
     * @param progress  The interface the netlet uses to notify the node of
     *     the progress of the termination. If the time is 0 then there is no
     *     reason to notify the node of progress.
     */
    public void terminate(long time, IShutdownProgress progress);

    /** Requests a service connection from this netlet.
     *
     * @param warrant   The warrant that was used to bind to this netlet.
     *     The implementation can assume that the node has checked the warrant
     *     by the time the implementation is called.
     * @param consumer  The consumer side of the connection. It is through
     *     this that the calling netlet offers its side of the two-way service
     *     connection.
     * @return  This side of the service connection. A null return is
     *     interpreted as a declined bind.
     */
    public IService bindService(Warrant warrant, IService consumer)
        throws ServiceBindException;

    /** Callback interface for the progress of a shutdown. This interface
     * will be given to a netlet by the node when it is requested to terminate.
     * The netlet can then use it inform the node of its shutdown progress.
     * Even though a node gives us a certain amount of time to shutdown
     * if we're seen to progressing well and we seem to need some extra time
     * then the node's administrator might be benevolent and give us a little
     * leeway.
     */
    public interface IShutdownProgress extends IRemote
    {
        /** Tell the node how we're progressing with our own shutdown.
         * @param progress  A percentage of the progress of the shutdown.
         */
        public void performed(double progress);
    }
}
```

4.3.5 jtrix.base.INode

This interface is passed to a netlet by the node on initialisation. It is used by the netlet to communicate with the node.

```
public interface INode extends IFacetCollection
{
    /** Request that the client netlet be terminated.
     */
    public void requestTermination();

    /** Ask the node to make a service connection. The node will instantiate a
     * service netlet. That netlet could be a proxy to the actual service.
     * @param warrant  The warrant that is used to bind to the service.
     * @param consumer  The consumer side of service. I.e. a service connection
     *     from the actual netlet making the request.
     * @return  The server side of the service.
     */
    public IService bindService(Warrant warrant, IService consumer)
        throws ServiceBindException;

    /** Create a service proxy, i.e. something with the same semantics
     * as a service connection but without binding a warrant.  This
     * proxy can be freely passed to other netlets.  Any facets passed
```

```
     * through the proxy will be dependent on the proxy, and will be
     * shutdown along with this proxy.  The proxy may have its own
     * thread pool.
     * @param impl  Something implementing the proxy
     * @param facet  The name of the interface that the proxy represents.
     *     The proxy object can then be cast into this.
     * @return  An interface to manage the service proxy, from which the
     *     proxied service itself can be obtained.
     */
    public IProxy createServiceProxy(IRemote impl, String facet);

    /** Tell the node how many concurrent threads it can start on the netlet's
     * behalf. For example, if we set the concurrency to 10 then the netlet
     * will only be able to accept 10 service requests, or 9 service requests
     * and 1 notification of termination. This does not affect how many
     * service requests the netlet can initiate. But if the netlet makes an
     * asynchronous call to a facet and the concurrency threshold is too low
     * then it may never receive the response. Since incoming calls are
     * otherwise out of the netlet's control, this method allows this resource
     * (threads) to have an upper limit.
     *
     * @param threads  Maximum number of threads the node can start for the
     *     netlet. Setting this to zero or less is silly, since it prevents
     *     the netlet from responding to any incoming request, including a
     *     notification of termination.
     *
     * @param thread_reuse  Can this netlet reuse threads? If a netlet invokes
     *     outwards synchronously then a causal incoming call was called
     *     synchronously then the original thread can be reused. Default is
     *     <tt>false</tt> (no thread reuse).
     */
    public void setConcurrency(int threads, boolean thread_reuse);
}
```

4.3.6 org.jtrix.base.IRemote

The mother of all mediated connections. Any facet must implement this. This is an empty interface; implementing it indicates to a node that a class can be proxied through a mediated service connection. When a facet is returned to a netlet through a service connection it is returned as an `IRemote` object; the netlet can then cast it into the desired type.

4.3.7 org.jtrix.base.IService

This interface is the means by which two netlets communicate with each other, for example to bind facets or request the relationship be terminated. Remember that a service connection is two-way—when we request an `IService` by presenting a warrant to the node we need to offer our own consumer `IService` to reciprocate.

```
public interface IService extends IFacetCollection
{
    /** Terminate this service connection. The node managing the connection
     * ensures that the connection is cut after the call is made and before
     * the implementing end receives the call.
```

```
     */
    public void terminate();
}
```

Chapter 5

The first netlet in detail

Note: *At the current time Jtrix Ltd is not running a public "hello world" service so this example cannot be run. However, the example has been left in for demonstration. Please do read it. Later we will see how to write our own service, so the lack of a public "hello world" service will not matter.*

This chapter looks at the original hello world program with our new found knowledge and vocabulary. We show the basics of a netlet, and how a netlet uses a service.

5.1 Hello world again

Our original hello world was a client of a hello world service. Here it is again, and we'll discuss it after. First, the imports:

```
package org.jtrix.project.helloworld;

import org.jtrix.base.*;
import org.jtrix.project.libjtrix.netlet.NullService;

/** Netlet which accesses a hello world service using a warrant
 * read from an incoming parameter.
 */
public class Hello1Client implements INetlet
{
```

The `initialise()` method does the main work:

```
    public byte[] initialise(INode node, Object param, byte[] unsigned)
        throws InitialiseException
    {
        try
```

```
        {
            // Get the warrant, use it bind the service, and then get the facet
            Warrant warrant = (Warrant)param;
            IService service = node.bindService(warrant, new NullService());
            String hf_name = IHelloFacet.class.getName();
            IHelloFacet facet = (IHelloFacet) service.bindFacet(hf_name);

            // Use the facet
            System.out.println(facet.getMessage());
            return null;
        }
        catch (Throwable e)
        {
            System.out.println("Sorry, no message available");
            e.printStackTrace();
            throw new InitialiseException(e.toString());
        }
    }
```

The class ends with minimal implementations of the rest of the `INetlet` methods:

```
    public void terminate(long date, INetlet.IShutdownProgress progress)
    {
        // Nothing to clean up when we terminate
    }

    public IService bindService(Warrant warrant, IService consumer)
        throws ServiceBindException
    {
        // This netlet offers no services to other netlets using warrants
        throw new ServiceBindException();
    }

    public String[] getFacets()
    {
        // This netlet offers no facets to the node
        return new String[0];
    }

    public IRemote bindFacet(String facet) throws FacetBindException
    {
        // The node shouldn't try to bind facets from here
        throw new FacetBindException();
    }
} // Hello1Client
```

5.1.1 Imports

The imports in this example are very simple.

The package `org.jtrix.base` carries the Jtrix definition: warrants, facets, netlet descriptors, and so on. Anything beyond this is extra. Almost every class in a Jtrix application will want to import this package.

The `NullService` class, imported next, is discussed later.

5.1.2 Implementing INetlet

Our class wants to be a netlet, so it must implement the `INetlet` interface. Every method in the class is actually part of `INetlet` or one of its super-classes, so this really is the smallest netlet we could write.

Of course a netlet is more than just a single class. Clearly we also need some classes and interfaces from `org.jtrix.base` and also `NullService` in Libjtrix. But it's this class which communicates with the node: when the netlet starts, it starts here; when the netlet stops it's this class which finds out first; and in general if there's any node/netlet communication this is where it takes place.

5.1.3 initialise()

This method is where the netlet begins execution—the node calls this method first.

There are three parameters. The first is the interface to the node; this lets us talk to it, and we will use it. The node is our runtime environment, so everything goes via that. We will use it to bind the hello world service.

The second parameter is an arbitrary parameter object (in our case, the warrant for the hello world service), and the third is rarely used and can be ignored for now.

We don't need to return anything particularly, so null will suffice.

If you're happy with that explanation you can skip to the next part, otherwise read on. . .

The second parameter is indeed some arbitrary object, in our case the warrant, and so it can be anything. But usually this object is a bean—i.e. just a holder of many useful objects we can pull out with "get" methods. This bean is also referred to as the "signed parameter bean" because it will have been carried inside the netlet descriptor, and the netlet descriptor can be digitally signed. There is no need for Jtrix to impose any other restrictions on this object; the developer is free here.

The third parameter is actually another arbitrary parameter—this is the unsigned data as discussed in Section 3.6. Because it is sent alongside the descriptor, being a data stream in its own right, it really is best described as a byte array.

5.1.4 Binding the service

Inside the `initialise()` method we do the main work—at least in this example.

The first thing is to take the arbitrary parameter and cast it into the warrant which we know it really is. (More robust code would test the object before casting.) Then we use it to bind the service.

This process of service binding is in many ways the heart of Jtrix. Since the node is our runtime environment we must ask it to bind the service for us. We have to give it our warrant, effectively saying "See this? I have the right to use this service. Go get it." The connection to the service goes into the `IService` variable called "service".

But while binding the service will give us a service connection, there's a bit more to it than that. The fact is, a service connection is two-way deal, and we can't get a connection from the hello service unless we give it a connection into us, too. So the `NullService` object from Libjtrix does just that. It's a simple service connection which offers absolutely nothing of interest. Yet by offering this we make a two-way service connection—the `NullService` class implements `IService`, and is given to the hello world service, just as we, the hello client, get its `IService` object which goes into the "service" variable.

Now we're bound to the service we can start to use it. We pull out a facet (i.e. an interface) by name. This is called binding, too.

And once we've got the facet we can use it, just as we can use any interface. In this case we use its `getMessage()` method and print out that message.

Of course, several things could go wrong along the way, so we do need to catch some exceptions. Among other things, the service might not be available. This might happen if it's died, or if our network connection has failed. Or perhaps the service is available, but our warrant is out of date. Or perhaps our `NullService` isn't good enough for it, and it wants us to provide a service which offers a credit card number.

And even if we do get to bind the service, perhaps binding the facet will fail. This will happen if the service doesn't offer the `IHelloFacet`—perhaps we picked up the wrong warrant and hence bound to the wrong service. Or perhaps the `IHelloFacet` is only available to those who offer a service with a credit card number. If we want to handle this situation elegantly then we can use the `getFacets()` method of the `IService` interface, and hence check to see if the `IHelloFacet` is supported.

Of course, these are all worst cases. If we're connected to the network properly, and we've remembered to put in the right warrant, and the warrant is for a fairly reliable service, then all will go well.

Now on to the rest of the `INetlet` methods.

5.1.5 terminate()

This method is how the node tells us we're about to get shut down. In this example we don't need to tidy anything up so we ignore the call.

But in general we can take whatever action is required. We get to know when we're due to terminate as a date. And we also get a callback interface so we can inform the node of our progress as we tidy up. The node's administrator might look at this and give us a bit more time if it looks like we're doing well but need a bit more leeway. Or they might notice we're stuck and shut us off straight away.

5.1.6 bindService()

This method is called when another netlet uses a warrant to bind to us. Of course, we, the hello world client, don't offer any warrant-based services, and in fact there won't be any warrants in the world which allow anyone to connect to us. So if, through some bizarre accident, this method should get called we must throw an exception.

But the very interesting thing about this `bindService()` method is that it is exactly this method inside the hello world service which gets called when we call `bindService()` in the `initialise()` method above. When we call `bindService()` with a warrant on our node the request goes right across to the hello world service, and its own `bindService()` method receives the request. All the networking is handled by the node (and there's quite a few things that happen in between) but all that's transparent to us. The point is, the `bindService()` method of `INetlet` is what gets called when another netlet uses a warrant to bind to us.

5.1.7 getFacets()

This `getFacets()` method allows us to tell the node what facets we can offer it. That is, it returns a list of netlet facets. In our case we don't offer any netlet facets and we say as much.

Note that this `getFacets()` method is what we, the netlet, offer our node. Do not confuse this what facets a service might offer. This `getFacets()` method is part of the `INetlet` interface, and as such talks about facets belonging to the netlet.

By contrast, look at the hello world service we use in the `initialise()` method above. That service presents itself to us as an `IService` interface and, as we mentioned there, we could have called a `getFacets()` method on that. But that would have told us what facets the service

offers. Netlets offer facets to nodes and they offer services to other netlets. Services offer facets to netlets.

One might think that we would want to have different method names to avoid confusion, such as `getNetletFacets()` and `getServiceFacets()`. However, the approach in use is logically correct. Netlets and services both offer collections of facets, so they use the same interface, `IFacetCollection`, with the same method. Other things are also facet collections (for example, nodes), so this interface is usefully general.

5.1.8 bindFacet()

This `bindFacet()` method allows the node to bind one of this netlet's facets. However, since we don't offer any facets to the node we should throw an exception if it tries to bind any.

This method is therefore a companion to the `getFacets()` method just discussed—a node should see what facets we offer using `getFacets()`, then bind one using `bindFacet()`.

5.1.9 Summary

That's all there is to this netlet. All the work is done in the `initialise()` method, and the rest is absolutely minimal. Over the next few chapters we'll expand on those other methods in various ways.

But before we continue, you may be wondering "Why is so much work happen in the `initialise()` method? Why not leave that method to perform very minimal initialisation and have something like a `run()` method in its own thread, executed by the node?" There are several parts the answer.

First, `initialise()` could start one or more threads of its own and then return. There's no restriction here; we're completely free.

Second, this is just an example, and we're doing so much in `initialise()` just to keep it simple. The `initialise()` method is supposed to be just setup and nothing more.

Third, netlets tend to be event-driven. When `initialise()` returns the netlet remains alive waiting for a call into one of its other methods. If another netlet wants to bind a service we offer then `bindService()` will get called; if the node wants to look at the facets we offer it, it will call `getFacets()`; and so on. Even though `initialise()` returns the netlet is still alive.

An example of a simpler `initialise()` and an event-driven call to `bindService()` can be found in Chapter 6. Threads are covered in great detail in Chapter 10.

Now on with the example.

5.2 IHelloFacet

Here's `IHelloFacet` again.

```
package org.jtrix.project.helloworld;

import org.jtrix.base.*;

public interface IHelloFacet extends IRemote
{
    public String getMessage();
}
```

Clearly there's not much to it. However, it's important that it implements `IRemote`, because this tells a node that it's okay to pass it through a mediator. All facets must implement `IRemote`.

5.3 Compiling the application

Compiling a Jtrix application is just a normal Java compile, but it's important to put our compiled classes, both `Hello1Client` and `IHelloFacet`, into a JAR.

In this case compilation requires two JARs to be on our classpath:

1. *jtrix.jar* because we're using `INetlet`, `IService`, etc;
2. *libjtrix.jar* because we're using `NullService`.

Then we'll put our newly-compiled classes in a JAR called *hello1.jar*. We don't even need to include a manifest, because we won't be running *hello1.jar* from the command line.

In general our Jtrix code will of course consist of many classes and perhaps other files, but after compilation they should all be bundled up into one or more JARs. Having the JARs enables our application to be moved from node to node.

5.4 Creating the netlet descriptor

We use *jtrixmaker* to create our netlet descriptor. Here's all the information we need to give it:

- What type of thing we're making. In this case, a netlet descriptor. (The program can make more other things, too.)
- What we want the resulting netlet descriptor to be called. In this case, *hello1-client.xml*.
- Where *jtrixmaker* can find the various JARs the netlet will use. Most of our Jtrix JARs are in */usr/lib/jtrix* (this might be different on your system), but *hello1.jar* is in our current directory (called "."), so we have to name that, too.
- Which JARs are needed to run the netlet. Certainly *hello1.jar* is needed, and also *libjtrix.jar*. But we don't have to name *jtrix.jar* because we can always assume it will be on whatever system we happen to be running on.
- What the main class is going to be, the one which is the `INetlet`. In this case it's `Hello1Client` which of course we have to specify with its full name.
- Any parameter we need to pass in to the netlet's `initialise()` method. In our case we need the warrant we put into the file *hello-warrant.xml*. If you don't already have it, get it from `http://www.jtrix.org/warrant/hello-warrant.xml`.

And with all that we can generate the netlet descriptor like this:

```
% ls
hello-warrant.xml  hello1.jar
% jtrixmaker -type netlet -outfile hello1-client.xml \
     -jardirs /usr/lib/jtrix . -jars libjtrix.jar hello1.jar \
     -classname org.jtrix.project.helloworld.Hello1Client \
     -param {warrant:hello-warrant.xml}
% ls
hello-warrant.xml  hello1-client.xml  hello1.jar
%
```

As we can see, the descriptor XML is created and we can move on to run it.

5.5 Running the netlet with Jnode

Recall that we run the netlet like this:

```
% jnode 201 -netlet-stdio hello1-client.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting hello1-client
netlet:201.0.1: Hello, world, and thank you for using our service
Bootstrap complete
^C

%
```

We run it in Jnode, which is just Nodality with a command line start-up. The arguments we've given it are:

1. An arbitrary node ID. This is required.
2. An option telling it to output standard I/O from netlets. This means we can see the `System.out.println()` from our netlet. Without this option set, we couldn't.
3. The descriptor for our netlet.

At the end, we need to hit Control-C to stop Jnode.

A word about the output. There are two kinds of output we see from Jnode. First, there is the output from the Jnode application itself. This is text which doesn't start with number sequences, such as "Bootstrap starting".

Second is the standard output from the netlets, from `System.out.println()` and so forth. Of course, this only appears if we've set the right option on the Jnode command line, which in this case we have. Each line of netlet output is preceded by a number sequence *a.b.c*—in this case 200.0.1. The first number is the node ID we gave Jnode. The second number is an "accounting group". Every netlet runs in an accounting group so the node can manage costs and (potentially) charge the right person. The third number is the netlet number within that accounting group.

Jnode buffers netlets' output, which means we may see output from different netlets in an order different from which they actually occurred. This doesn't affect us this time, but will when we run several netlets from the Jnode command line.

Meanwhile, Jnode doesn't buffer its own output. Again, this means its output may appear in a sequence slightly different from what actually happens in relation to its netlets. For example, netlet initialisation occurs during Jnode's bootstrap sequence, and our netlet prints output during this, but we may see "Bootstrap complete" before we see the netlet's output. Or we may see it the other way round.

5.6 Summary

In this chapter we've spent some time on the practical side of writing, compiling and running netlets. Our netlet used a warrant to access a real service, and it used a facet from that service.

In the chapters that follow we'll write a Jtrix service, then write various other netlets to explore other aspects of nodes and netlets.

Chapter 6

Writing a service

Now that we've written a client netlet we can look again at services with the aim of writing one in this chapter.

6.1 Recap on services

In our first netlet example, `Hello1Client`, we used a warrant to access a service. That netlet, and indeed any netlet, uses the `bindService()` method to connect to such a service.

Now we know that any such service is accessed via another netlet. That other netlet, the access point netlet, downloads and presents facets (such as `IHelloFacet`) to the client netlet.

In our example the facet supplied a not-very-exciting hello message, and it was unclear whether that message came directly from the access point netlet or from a central server, and just retrieved by the access point netlet. But in fact it didn't matter. The access point netlet itself was not accessible to our client netlet—only its facet was.

And that's really the point about services: all the client deals with is the service interface, `IService`, and everything else comes from there. The access point netlet is irrelevant to the client because it just deals with the service interface.

Nevertheless, that access point netlet is part of the service provided (supplying a hello message). Writing a service involves, at the very least, having an access point to supply the service. Here are some example hello world services and how they might work:

- Hello world service 1. The access point sends a request back to a central server to get the message and returns it to the client. The central message server is clearly a single point of failure here.
- Hello world service 2. The access point downloads with a list of available message servers and asks one of them for the message. Whenever one message server dies the others create a new one. They update the current list carried by all current access points and ensure that future access points all start with this latest list.
- Hello world service 3. An extension of the previous example, but here the service administrator can connect and change the message whenever they wish. When they connect they get an access netlet, of course, and it again connects to one of the message servers. When they update the message on the one message server it percolates that change to all the others. That way they are all kept up to date.
- Hello world service 4. The access point carries the message itself. It doesn't need to worry about any message servers; they don't exist. This service consists only of access points which don't communicate with each other. The message cannot be updated on the fly.

Hello world service 3 might seem wildly ambitious, but in fact that's just the kind of thing the Beatrix framework provides. If you've downloaded the hello world warrant from the Jtrix.org site, you're using Hello world service 2, which is only a small step away from it. It was written with just seven classes within Beatrix. See *Programming with Jtrix: The Beatrix application framework*[1] for more details.

However, the humbler Hello world service 4 is still a legitimate Jtrix service. It's not as wild, but it's a good solid start. That's what we'll write now.

6.2 Discussion of our code

Our service will be a small access point netlet that implements an `IHelloFacet`.

Also, it's important that this new access point netlet is going to be completely compatible with our previous client netlets: anything that binds an `IHelloFacet` should be able to use this service. In fact, we'll run this service with the `Hello1Client` we've already written.

[1] `http://www.jtrix.org/download/5/misc/jtrix-beatrix-programming.pdf`

6.2.1 Implementing IHelloFacet

As already mentioned, our service will have to implement the `IHelloFacet`. It's not complicated.

It's worth taking this opportunity to talk again about security. (You can skip this bit if you want.) Although the client netlet uses this interface it doesn't use any of our implementation. Our implementation of `IHelloFacet` stays entirely within our codebase, thanks to the mediator. Whenever the client invokes `getMessage()` the call comes back to our access point netlet which deals with it, so that means the service is only usable as long as our acess point netlet is alive, and we cannot use nasty techniques to compromise the client: we're completely isolated from it.

6.2.2 Providing an IService

A big point of interest is the `bindService()` method of `INetlet`. We'll ignore its parameters for now, but let's have a look at `IService` itself (Section 4.3.7). It's just a collection of facets which can be terminated together. So any `IService` interface has to implement the following methods. Note these are described from an implementor's point of view, not a client's point of view:

- `getFacets()` needs to return a list of those facets this service connection supports. In our case, just the one.
- `bindFacet()`. Called by the client when it wants to bind a particular facet. If it doesn't request the `IHelloFacet` we should throw an exception.
- `terminate()` needs to respond to the connection with the client being severed. The node actually terminates the connection, possibly on the request of the client. When we receive a call here it's just a notification.

Notice that these three methods appear not only in the `IService` interface, but also in the `INetlet` interface. However, let's not confuse them. Each method does the same sort of thing in both interfaces, but they're used in completely different contexts. The `INetlet` interface is presented to a node, so it can talk to the netlet, and so there the methods offer facets to the node, and respond to netlet termination from the node. But the `IService` interface is presented to another netlet, making a service connection, and so there the methods offer facets to the other netlet and respond to a termination of the service connection.

If you didn't get all that, it's worth reviewing. An `IService` interface represents a service connection to another netlet. The `INetlet` interface is how the netlet presents itself to its node.

In particular, a call to `INetlet.terminate()` means "This netlet is about to terminate", but a call to `IService.terminate()` means "The other netlet has terminated its connection to you". In that latter case perhaps other netlets are still using us and we'd want to keep on running.

But in our current little example, if the client netlet terminates its connection with us then we no longer serve a purpose and our access point netlet will ask the node to put us out of our misery.

6.2.3 FacetHandle

In this example we will finally meet the `FacetHandle` class mentioned in Section 4.3.1. It's really nothing to worry about although it can be a bit confusing to Jtrix beginners. For the moment, let's just keep a lookout for it and we'll discuss it after.

6.2.4 How we'll use our service

How are we actually going to persuade our client netlet to use our new service? Well, access to any service is via a warrant, so we'll have to create a new warrant.

The last warrant we used, in Chapter 5, was downloaded from the Jtrix.org site. That was a warrant for the hello world service provided by Jtrix.org. If we give the same client netlet a different warrant it can use a different service.

What's the key information we need to put in the warrant? A warrant tells the node how to bind the access point netlet. That means it contains either (a) a netlet descriptor, describing the access point netlet, or (b) a reference to where the netlet descriptor can be downloaded from. (See Figure 2.4.) We'll use option (a). We'll embed a descriptor of the access point netlet inside the warrant.

So the client netlet will use this new warrant. It will present it to the node via the node's `bindService()` method, the node will look in the warrant to see how to bind the access point netlet, it will see the netlet descriptor embedded in there and it will use that descriptor to create the access point netlet.

The netlet descriptor must also contain all the JARs for its codebase, or else URLs saying where they can be found.

Now let's have a look at the netlet code. . .

6.3 HelloServer

We'll look at our code and discuss it as we go along.

6.3.1 The main netlet methods

We begin our class as follows:

```
package org.jtrix.project.helloworld;

import org.jtrix.base.*;

/** Netlet which provides a Hello World service in a very simple way.
 */
public class HelloServer implements INetlet
{
    private static final String _facet_name = IHelloFacet.class.getName();
    private INode _node;
```

We declare two private variables, just as placeholders for later.

Since this class is a netlet we must implement the INetlet methods:

```
    public byte[] initialise(INode node, Object param, byte[] unsigned)
        throws InitialiseException
    {
        _node = node;
        System.out.println("Server started");
        return null;
    }

    public void terminate(long date, INetlet.IShutdownProgress progress)
    {
        // Nothing to clean up when we terminate
    }

    /** Provide the service requested by the client.
     * @param warrant   The warrant which the client used for binding
     * @param consumer  The client's reciprocating service interface.
     * @return  A service connection.
     */
    public IService bindService(Warrant warrant, IService consumer)
        throws ServiceBindException
    {
        return new HelloService();
    }

    public String[] getFacets()
    {
        // We offer only the IHelloFacet
        return new String[]{ _facet_name };
    }

    public IRemote bindFacet(String facet) throws FacetBindException
    {
        // The node shouldn't try to bind facets from here
        throw new FacetBindException();
    }
```

The `initialise()` method doesn't do much except save a reference to the node. As mentioned above, this is so we can ask it to terminate us if the connection to the client is severed. The `terminate()` method does even less.

We don't offer any facets to the node (we do offer facets to other netlets, but not to the node) hence the contents of `getFacets()` and `bindFacet()`.

But of most interest is `bindService()` which is called when another netlet tries to bind a service from us. This will be the client netlet asking for the hello world service. We return a new `HelloService` object which implements `IService`, as required, and which we'll get to shortly.

6.3.2 Implementing IHelloFacet

We know the client is going to use the `IHelloFacet`, because that's part of what we offer, so we need to provide an implementation. Here it is, a private class within our netlet:

```
/** A simple implementation of IHelloFacet
 */
private class HelloFacet implements IHelloFacet
{
    public String getMessage()
    {
        return "Hello, world";
    }
}
```

6.3.3 Providing the service

Finally here's the implementation of `HelloService` which actually provides the service to the client netlet. Recall this is what was returned from the `bindService()` method above:

```
/** This is the real implementation of the hello world service.  It's
 * handed out only through the bindService() method of the public
 * (netlet) class.
 */
private class HelloService implements IService
{
    /** Respond to the service connection failing. If it does fail
     * this netlet is useless, so we terminate ourselves.
     */
    public void terminate()
    {
        System.out.println("Service terminated, terminate ourselves");
        _node.requestTermination();
    }
```

```
        /** See what facets this service offers the binding (consumer) netlet.
         */
        public String[] getFacets()
        {
            return new String[] { _facet_name };
        }

        /** Allows the consumer netlet to bind a specific facet.
         * @param facet  The name of the facet to bind.
         * @return  An interface to the requested facet.
         */
        public IRemote bindFacet(String facet) throws FacetBindException
        {
            System.out.println("Instantiating "+facet);

            if(!facet.equals(_facet_name))
            {
                throw new FacetBindException();
            }

            System.out.println("Facet instantiated");
            return new FacetHandle(new HelloFacet(), facet);
        }

    } // HelloService

} // HelloServer
```

Notice this class implements all the methods we mentioned in our earlier discussion: `getFacets()`, `bindFacets()` and `terminate()`.

The `terminate()` method responds to the client netlet severing the service connection, and does so by self-destructing the netlet.

The `getFacets()` methods tells the client netlet what facets it's offering. In this case, just the one, the `IHelloFacet`.

But most interesting is the `bindFacet()` method, which binds any facet requested by the client. Of course, the first thing it does is to check the client is asking for a legitimate facet—if it's not asking for the one facet it offers then it throws an exception.

But notice how it returns the `HelloFacet` object, the implementation of `IHelloFacet`. In a simpler world it would just return a `new HelloFacet()` but instead it wraps it in a `FacetHandle`. If you want a simple view of the world, just remember that whenever a netlet needs to return a facet it should wrap it in a `FacetHandle`, giving the object and the facet interface name. If you want to know a bit more, read on...

6.3.4 Why we use FacetHandle

Whenever an object is passed from one netlet to another, or from a node to a netlet, or vice versa, it goes via the mediator.

The mediator's job is to ensure Jtrix components never share code, so the object isn't actually passed through. Instead the mediator creates a proxy of the object and passes that.

The problem is, it isn't always easy for the mediator to work out which bits of the object are important in the proxying, and which bits aren't. It has a real problem when we have to return a facet. So when we return out `IHelloFacet` we have to explicitly say "This is the object and this is the interface name" (the two parameters of the `FacetHandle` constructor).

But the rest of the time we never have to worry about it. For example, the return value of `getMessage()` is a `String`, and we don't need to wrap that, even though it goes through the mediator (just like all things from one netlet to another pass through the mediator).

So that's the rule: when returning a facet, wrap it in a `FacetHandle`.

6.4 Compiling the code

When we compile the code we need to include *jtrix.jar* on our classpath, which contains `org.jtrix.base`.

Then we put our newly-compiled code into a JAR called *helloserver.jar*. That JAR should contain the class files *HelloServer.class*, *HelloServer$HelloFacet.class*, *HelloServer$HelloService.class* and *IHelloFacet.class*, all of which make it needs to run.

6.5 Creating the XML files

Our warrant has a netlet descriptor embedded in it, so we'll create the descriptor first and then the warrant:

```
% ls
hello1.jar  helloserver.jar
% jtrixmaker -type netlet -outfile hello-server.xml \
     -jardirs . -jars helloserver.jar \
     -classname org.jtrix.project.helloworld.HelloServer
% ls
hello-server.xml  hello1.jar  helloserver.jar
% jtrixmaker -type warrant -descriptor-in hello-server.xml \
     -outfile hello-local-warrant.xml
% ls
hello-local-warrant.xml  hello-server.xml  hello1.jar  helloserver.jar
%
```

The first call to *jtrixmaker* creates the descriptor for the netlet we've just written. We give it this information:

- What type of file we're creating, in this case a netlet descriptor.
- What file it should be output to.
- What directories to look in for the JARs. In this case, just the current directory.
- What JARs need to go in the netlet descriptor. In this case, just *helloserver.jar*. Of course, the code also makes use of *jtrix.jar*, but we are guaranteed this will always be available on any Jtrix node we run on, so we don't need to carry it with us.
- The class which implements the `INetlet` interface.

Once we've got the descriptor, the second call to *jtrixmaker* creates the warrant. We give it this information:

- What type of file we're creating, in this case a warrant.
- What descriptor should go in the warrant. In this case the file *hello-server.xml* we've just created. Other options to *jtrixmaker* allow us to point to a SAS, instead, which will serve the descriptor.
- What the resulting warrant file should be called.

Now we can create the netlet descriptor for the client netlet. Why should we do this, you may ask, after all, aren't we using the same client netlet from before? The answer is Yes, but look back to where we created that descriptor (Section 5.4). Notice that we put the warrant inside it as a parameter. So now we must recreate it, this time using our new warrant as the parameter:

```
% ls
hello-local-warrant.xml  hello-server.xml  hello1.jar  helloserver.jar
% jtrixmaker -type netlet -outfile hello1-client.xml \
     -jardirs /usr/lib/jtrix . -jars libjtrix.jar hello1.jar \
     -classname org.jtrix.project.helloworld.Hello1Client \
     -param {warrant:hello-local-warrant.xml}
% ls
hello-local-warrant.xml  hello1-client.xml  helloserver.jar
hello-server.xml         hello1.jar
%
```

In a future chapter (Chapter 8) we'll write a client netlet that reads in its warrant parameter from the command line. But that's for later. For now, let's get on and run our example.

6.6 Running the code

Running the netlet is exactly the same as before. We start a node with our client netlet. However, this time the output from the service is slightly different, including the message:

```
% jnode 211 -netlet-stdio hello1-client.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting hello1-client
Bootstrap complete
netlet:211.0.2: Server started
netlet:211.0.2: Instantiating org.jtrix.project.helloworld.IHelloFacet
netlet:211.0.2: Facet instantiated
netlet:211.0.1: Hello, world
^C

%
```

At the end of this Jnode continues running, so we have to hit Control-C to stop it and get our prompt back.

Just as before, *jnode*'s first argument has to be some arbitrary node ID, and we again follow it with the option to output netlet standard I/O. Then we name our client netlet.

Jnode runs the netlet, which causes our new access point netlet to start up. It gives a short commentary on its progress and the first netlet announces the message it retrieved.

And let's remind ourselves about Jnode's output, which this time is a bit different. For one thing, there are two netlets, and they have output 211.0.1 (for the first netlet, the client) and 211.0.2 (for the second netlet, the access point and server).

For another thing, we cannot expect each netlet's output to appear in the correct order with respect to the other netlet and Jnode itself. The buffering we mentioned with the last example is the cause of this.

6.7 More discussion on services

Now we've written a simple service, and perhaps we can see better its shortcomings and where things might go from here.

6.7.1 Access points' communication

The main thing missing is that the access point netlet, the one we just wrote to give the client access to the service, is entirely self-contained

and isn't part of a distributed application. This isn't the way things need to be; it's just we chose to write a very simple example.

In practice a netlet can happily make network connections (see Chapter 11 and the document *Programming with Jtrix: The Beatrix application framework*. In fact, an access point netlet will want to keep up to date with several other netlets as we described in the Hello world service 3 earlier. This federated approach is what Beatrix is really good at.

It shouldn't be surprising that helping a netlet communicate with its peers is not part of the Jtrix core and is instead part of Beatrix, an optional extra. This is because Jtrix doesn't impose any networking standards of its own. Your application is free to use whatever is best for it: SOAP, RMI, HTTP or anything else.

6.7.2 More flexible code: SAS

The warrant to our local hello server, the *hello-local-warrant.xml*, carries its own descriptor which carries its own codebase. This is a disadvantage for two reasons.

First, and most obviously, the descriptor's codebase contains encoded JARs which makes it huge. This is even more obvious for the netlet descriptor of the client which contains *libjtrix.jar* which is very large. And this isn't even a large application.

Second, the client netlet may carry the warrant for some time before it uses it, and during this time the code for the service may get out of date. In fact in general a service will always want to deliver the latest code, regardless of how old is the client's right to use that service—i.e. regardless of how old their warrant is.

The solution to both these problems is a SAS.

First, JAR files can be uploaded into a Web server and referenced by URL. *jtrixmaker* contains options to reference JARs by URL and not have them embedded in a netlet descriptor. See the options *–jar-bases*, *–jar-urls* and *–embedded* in Appendix B. A SAS acts as a simple Web server, allowing JARs to be uploaded and referenced by URL.

Second, SAS also allows netlet descriptors to be uploaded and referenced by URL, so these do not have to be carried in the warrant, but only referenced. The downloaded *hello-warrant.xml* contains an example of these so-called bind servers, because the URLs say where the service can be bound from. See the option *–bind-urls* of *jtrixmaker* in Appendix B.

This latter feature of a SAS, offering netlet descriptors, is its main feature. It's a bit more than just an HTTP download because the SAS needs to send a bit of header information with the netlet descriptor.

Once we have our netlet descriptors and JARs uploaded into a SAS it also helps us update them as necessary, as well as sending selected appropriate netlet descriptors to different clients, depending on their needs.

Thus a fully-fledged Jtrix service will make use of a SAS, just to help it manage its distribution more easily. Again, Beatrix aids in this a lot.

6.7.3 Expanding ourselves: the hosting service

Another thing our simple service doesn't do—because it's just not needed—is expand and contract as needed.

Controlling the number of access points is a nonsense idea, because access points are created whenever a client binds a warrant to access our service, and that's out of our control. But controlling the number of core netlets which run our service does make sense.

Looking back at our example Hello world service 3 we talked about a number of message servers which the access point netlets queried to get the message. Clearly if our hello world service was in high demand then we'd want more message servers to cope with demand, perhaps located in strategic locations around the world to get closer to the clients.

This is what a hosting service is for. Access to a hosting service allows us to send netlet descriptors onto its nodes for those nodes to run them. Therefore a fully-fledged Jtrix service will make use of several hosting services to spread itself out as needed. Again, Beatrix contains tools to make this relatively easy. Beatrix keeps several message servers running for Jtrix.org's hello world service, and all the service implementation worries about is how many it needs. Whenever a message server gets killed Beatrix notices and asks a hosting service to start another in its place.

6.8 Summary

In this chapter we've seen how to write a Jtrix service and discussed in some detail about what else Jtrix services might do.

At the time of writing (January 2002) there is a lot of hype about "Web Services", which tends to mean remote procedure calls via SOAP. We've

seen there's much more to providing Web-based services than SOAP remote procedure calls, and Jtrix provides complete flexibility all round, allowing most if not all related technologies as needed.

In the next few chapters we'll look some more interesting cases of what nodes and netlets can do. Meanwhile you can read more about Beatrix in *Programming with Jtrix: The Beatrix application framework* at `http://www.jtrix.org/download/5/misc/jtrix-beatrix-programming.pdf`.

Chapter 7

Using node facets

Here is another simple netlet, this time to demonstrate node facets.

7.1 Discussion

Recall from Section 3.4 that nodes can offer facets, and these are called node facets. Not only that, Nodality allows a netlet to add facets to Nodality's existing collection. In this example we explore this.

If you like, you don't need to concentrate too hard on the discussion that follows. Instead you might prefer to skim it and get onto the code; then, when you come across something you don't understand, refer back to here.

7.1.1 The netlets

Our example consists of two netlets. The first simply uses a node facet. This client netlet is even simpler than our previous example because no warrant is needed to access a node facet; if the netlet is running on that node it just asks the node for it. The facet we ask for is, of course, the `IHelloFacet`.

Now you might be thinking "Why would a node offer an `IHelloFacet`? A node has the potential to offer many useful features, but offering a facet to get a simple hello world message is surely arbitrary and a waste of time." This is fair enough. The fact is the node doesn't offer `IHelloFacet` among its default facet collection, but this collection can be extended and that's where the second netlet comes in.

The second netlet provides the `IHelloFacet`. (Or, more correctly, it provides an object with implements `IHelloFacet`, because `IHelloFacet` is an interface.) This more interesting. It doesn't directly provide an `IHelloFacet`—what it actually does is tell the node that it *can* provide an `IHelloFacet` if needed.

One of Nodality's own facets is an admin facet called `INodeAdminFacet`, and one of `INodeAdminFacet`'s features is the ability to add to Nodality's existing facet collection. The provider netlet supplies an object which can produce an `IHelloFacet`.

That's a procedure worth repeating: Nodality's `INodeAdminFacet` is given an object which, when asked, gives an `IHelloFacet`.

Note that `INodeAdminFacet` is certainly not given an `IHelloFacet` itself. The `IHelloFacet` may never be requested. Or it may be requested several times by different netlets, and we might not want to give the same implementation on every request. So the `INodeAdminFacet` is given a object from which it can get an `IHelloFacet` if ever it's needed.

7.1.2 Why do we have node facets?

A brief detour to talk about motivation. This example is all very well, but why, in general, do we have the concet of node facets?

The answer is that it makes it very easy to extend the node's functionality. Rather than extending the node's own code it allows us to write a netlet to do this, acting as plugin. Other netlets can use this node facet. Further, the providing netlet can impose permission-based access to help control use.

7.1.3 The classes

These are the classes we implement:

Hello2Client The client netlet. This implements `INetlet`. It will ask the node for the `IHelloFacet`.

Hello2Provider The provider netlet. This also implements `INetlet`. It will tell the node it can supply an `IHelloFacet`. It needs to execute before the client netlet, otherwise the node won't be able to handle the client's request.

FacetProvider This is the object given to the node which it knows can, if requested, produce an `IHelloFacet`. It is an inner class of `Hello2Provider`. It implements an interface called `INodeFacetCollection`, which is simply what Nodality insists such an object implements.

MessageGiver This is the implementation of `IHelloFacet`. Whenever the `FacetProvider` is asked to provide an `IHelloFacet` it constructs a new one of these.

IHelloFacet As before.

7.1.4 Security

A few words on security. Jtrix is very hot on security, and among other things this means that no netlet have alien code running in its own class space. And that goes for the node, too—the node should never have to run alien code in its class space.

What this means for us is that our provider netlet always runs the implementation of the `IHelloFacet`. So that provider netlet has to stay alive for as long as the client netlet wants to use it. When the client netlet asks the node for the `IHelloFacet` and the node goes to the provider netlet to get it, it stays entirely with the provider's space. Thus both the node and the client netlet are protected from any rouge actions the provider might perform.

7.1.5 The FacetHandle class

In this example we will again meet the `FacetHandle` class mentioned in Section 4.3.1. Recall that a `FacetHandle` is just a facet implementation plus the facet name. When the `FacetProvider` is asked for an `IHelloFacet` it actually returns a `MessageGiver` together with the name of the `IHelloFacet` interface, both bundled together as a `FacetHandle`.

When the `FacetProvider` class returns the `MessageGiver` it is returning an object which implements an `IHelloFacet`. What the client netlet gets is the `IHelloFacet` as requested, but it's not the `MessageGiver`—it's a proxy object which implements `IHelloFacet` and which links back to the `MessageGiver`. This is all for the security we talked about, because every netlet must be protected from alien code. Since the node creates the proxy and mediates the client/provider link it must be told which bits of the `MessageGiver` are important. So by returning a `FacetHandle`—which is just the facet implementation plus the facet name—the node can see that it's what interface its proxy must implement, and what implemenation to link back to.

7.2 Hello2Client

The introduction to `Hello2Client` is simple. It only uses classes from `org.jtrix.base`.

```
package org.jtrix.project.helloworld;

import org.jtrix.base.*;

/** Netlet to get a message from a node facet and output it.
 */

public class Hello2Client implements INetlet
{
```

The netlet's `initialise()` method only uses the first of its three parameters, which is the link back to the node. It uses this to bind the node facet and then it uses it. Notice that to bind a node facet we just take the `INode` interface and bind. Compare this to the previous example where we bound a facet from a service.

```
    /**
     * Called by the node on binding the netlet. Part of the INetlet
     * interface. This method will actually bind the facet,
     * then get the message.
     *
     * @param node      An interface through which we can talk to our node.
     * @param bean      A parameter bean passed in the descriptor. Not used.
     * @param unsigned  Some arbituary data from the bind server.
     * @return          Arbitrary netlet-specific data.
     */
    public byte[] initialise(INode node, Object bean, byte[] unsigned)
        throws InitialiseException
    {
        try
        {
            String facet_name = IHelloFacet.class.getName();
            IHelloFacet facet = (IHelloFacet)(node.bindFacet(facet_name));
            System.out.println(facet.getMessage());
        }
        catch(Exception e)
        {
            System.out.println("Sorry, couldn't get the message");
            e.printStackTrace();
            throw new InitialiseException(e.toString());
        }

        return null;
    }
```

And finally the rest of the `INetlet` methods are just as for our previous example, since `Hello2Client` does nothing more:

```
    /** No clean-up needed on terminate.
     */
    public void terminate(long date, INetlet.IShutdownProgress progress)
    {
        // No clean-up necessary
```

```
    }

    /** Called by another netlet to bind this netlet's services. We don't
     * have any.
     */
    public IService bindService(Warrant warrant, IService consumer)
        throws ServiceBindException
    {
        throw new ServiceBindException();
    }

    /** Called by the node to see what facets we can offer it.
     * @return  An empty array, since we don't offer any facets.
     */
    public String[] getFacets()
    {
        return new String[0];
    }

    /** Allows the node to bind any facets we can offer it.
     * @throws FacetBindException  Thrown every time since we offer no facets.
     */
    public IRemote bindFacet(String facet) throws FacetBindException
    {
        throw new FacetBindException();
    }

} // Hello2Client
```

And that's the end of the netlet. It binds and uses a node facet during its initialisation, and nothing more.

7.3 Hello2Provider

The introduction here is again quite straightforward. However, we do need to import a couple of Nodality-specific interfaces. We also need to use a class called `NetletID`, which we'll discuss when we get to it.

```
package org.jtrix.project.helloworld;

import org.jtrix.base.*;
import org.jtrix.project.nodality.facet.INodeAdminFacet;
import org.jtrix.project.nodality.facet.INodeFacetCollection;
import org.jtrix.project.nodality.facet.NetletID;

/** Netlet to provide an IHelloFacet to the node, so it can then offer it
 * to other netlets.
 */

public class Hello2Provider implements INetlet
{
```

The `initialise()` method is where we offer our `FacetProvider`. Have a look at it first; we'll discuss it afterwards.

```
    /**
     * Called by the node to initialise the netlet, this method adds
```

```
     * its facets to the node's current collection.<p>
     *
     * The procedure is: (1) Bind the node's administration facet.
     * (2) Use this to give our FacetProvider object.
     */
    public byte[] initialise(INode node, Object bean, byte[] unsigned)
        throws InitialiseException
    {
        try
        {
            String na_name = INodeAdminFacet.class.getName();
            INodeAdminFacet na = (INodeAdminFacet)node.bindFacet(na_name);
            na.addNodeFacets(new FacetProvider(), null);
        }
        catch (Exception e)
        {
            System.out.println("Netlet failed to start");
            e.printStackTrace();
            throw new InitialiseException(e.toString());
        }

        System.out.println("Netlet started");
        return null;
    }
```

Again we bind a node facet, this time the `INodeAdminFacet`, through which we add our node facet which the client will use. The `addNodeFacet()` method has two parameters:

1. The object which can, if required, produce our facets.
2. Any permissions a netlet needs to access the facets. Null means no permissions are required—any netlet can access our facet. We won't go into this subject any more here.

Our other `INetlet` methods are the usual do-nothing implementations:

```
    public void terminate(long date, INetlet.IShutdownProgress progress)
    {
        // Nothing to clean up when we terminate
        System.out.println("Netlet stopped");
    }

    public IService bindService(Warrant warrant, IService consumer)
        throws ServiceBindException
    {
        throw new ServiceBindException();
    }

    public String[] getFacets()
    {
        return new String[0];
    }

    public IRemote bindFacet(String facet) throws FacetBindException
    {
        throw new FacetBindException();
    }
```

Here's how we implement the `FacetProvider`. This is the one we give to Nodality which can, if needed, produce the facets. Again, let's look at it first and discuss it after.

```
/**
 * This provides a hello world facet to the node, so the node can
 * offer a hello world service to netlets as part of its hosting.
 * This class is an INodeFacetCollection, which means it must be able to
 * say what facets it offers and then bind them.
 */
private class FacetProvider implements INodeFacetCollection
{
    private final String _facet_name = IHelloFacet.class.getName();

    /** Say what facets we offer (only the one).
     */
    public String[] getFacets()
    {
        return new String[]{ _facet_name };
    }

    /** Bind the named node facet, as requested by a particular netlet.
     * @param id  Some Nodality-specific ID of the netlet requesting the
     *            node facet.
     * @param facet  The name of the facet being requested.
     */
    public IRemote bindNodeFacet(NetletID id, String facet)
        throws FacetBindException
    {
        System.out.println("Instantiating "+facet);

        if(!facet.equals(_facet_name))
        {
            throw new FacetBindException();
        }

        System.out.println("Facet instantiated");

        return new FacetHandle(new MessageGiver(), facet);
    }

} // FacetProvider
```

As mentioned, this is an `INodeFacetProvider`, which is simply what Nodality requires of any such object, and that just means it implements the methods called `getFacets()` and `bindNodeFacet()`.

The method `getFacets()` just returns an array saying which facets we're offering. In this case, just the one, the `IHelloFacet`.

The method `bindNodeFacet()` is what the node calls whenever that facet is needed. The first parameter is just an identifier of the consumer netlet; this is the `NetletID` class we had to import earlier. The second parameter is the name of the facet it wants to bind. The method can't cope with a request for anything but the `IHelloFacet`. But if that is what's requested it can return the implementation of `IHelloFacet`—a `MessageGiver`. As mentioned above, it must wrap the `MessageGiver` in a `FacetHandle`, so the node can proxy it properly.

Finally, here's the `MessageGiver`:

```
    /** The implementation of the IHelloFacet.
    */
    private class MessageGiver implements IHelloFacet
    {
        public String getMessage()
        {
            return "Hello, world";
        }
    } // MessageGiver

} // Hello2Provider
```

7.4 Compiling the application

To compile our three classes (`Hello2Client`, `Hello2Provider` and `IHelloFacet`) we need these JARs on our classpath:

1. *jtrix.jar* as always, as it contains `INetlet`, `IService` and everything else that defines Jtrix.
2. *nodality_facet.jar* which contains the extras that we imported to `Hello2Provider`.

Then we'll bundle all our compiled *.class* files into a single JAR called *hello2.jar*, and again we don't even need a manifest.

7.5 Creating the netlet descriptors

We need two netlet descriptors for this example, one for the client netlet and one for the provider netlet.

For the creating client netlet descriptor we specify the following:

- What type of thing we're making—a netlet descriptor.
- What the resulting netlet descriptor will be called—in our case, *hello2-client.xml*.
- Where to find our JARs—on our system it's */usr/lib/jtrix* and the current directory.
- What JARs the netlet needs. Only one, in this case, which is *hello2.jar*. It contains the `Hello2Client` and the `IHelloFacet`. But it doesn't need anything else, because although it does use `INetlet`, `IService`, etc from *jtrix.jar* it can always assume that *jtrix.jar* will be available for it.

- The class name of the netlet—`Hello2Client`.

Meanwhile, the netlet descriptor for `Hello2Provider` is largely the same. It differs only in these respects:

- The resulting netlet descriptor has a different filename. We choose *hello2-provider.xml*.
- It needs *hello2.jar* as with the client, but this netlet also needs *nodality_facet.jar* because that contains the `INodeAdminFacet` and others through which it offers its `IHelloFacet`.
- The class name for this netlet is `Hello2Provider`.

From all this we get the following sequence:

```
% ls
hello2.jar
% jtrixmaker -type netlet -outfile hello2-client.xml \
     -jardirs /usr/lib/jtrix . -jars hello2.jar \
     -classname org.jtrix.project.helloworld.Hello2Client
% jtrixmaker -type netlet -outfile hello2-provider.xml \
     -jardirs /usr/lib/jtrix . -jars nodality_facet.jar hello2.jar \
     -classname org.jtrix.project.helloworld.Hello2Provider
% ls
hello2-client.xml  hello2-provider.xml hello2.jar
%
```

The two descriptors that come out of this allow us to run them in a node.

7.6 Running the example with Jnode

Here's how we run the two netlets in a new Jnode node, with the output:

```
% jnode 202 -netlet-stdio hello2-provider.xml hello2-client.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting hello2-provider
netlet:202.0.1: Netlet started
starting hello2-client
Bootstrap complete
netlet:202.0.1: Instantiating org.jtrix.project.helloworld.IHelloFacet
netlet:202.0.1: Facet instantiated
netlet:202.0.2: Hello, world
^C

%
```

At the end of this Jnode continues running, so we have to hit Control-C to stop it and get our prompt back.

As always, Jnode's first argument is some arbitrary node ID, and we follow it with the usual option to output netlet standard I/O. Then we name the two netlet descriptors we've just created.

Jnode runs the first netlet, and waits til that's finished booting before running the second one. So we can be quite sure Nodality has been told about our `FacetProvider` by the time the second netlet, the client, asks for the facet.

The output comes from both netlets this time, as well as from Jnode itself. The netlets have output 202.0.1 (for the first netlet, the provider) and 202.0.2 (for the second netlet, the client).

As always, Jnode's buffering means the netlets' output relative to each other does not necessarily line up, but of course each netlet's output does appear in the order in which its own events occur.

7.7 Summary

We've seen here that facets come from sources other than services, how Jnode behaves when it's running two netlets, and we've also seen a little bit of Nodality and its extensibility. This example has also shown us the value of separating interfaces and implementations (which Jtrix is very hot on) because we only needed a small JAR of Nodality's facets to work with it, and didn't need to import a large JAR containing the whole of Nodality itself.

Chapter 8

A bootstrap netlet

In our next hello world example we write a bootstrap netlet, an idea we first mentioned in Section 2.6. Our bootstrap netlet will read in a warrant and use it to access a hello service. Its action is very similar to our original `Hello1Client`, with the major difference being how it gets the warrant. The bootstrap netlet idea is used by Jtrix.org's hosting service, its SAS, and others, and thus is an important concept with a long a distinguished history. This example also involves creating a netlet facet for the first time.

8.1 Discussion

A brief refresher (and some more detail) on bootstrap netlets, then a discussion of the code we're going to write.

8.1.1 Bootstrap netlets

A bootstrap netlet is special to Jnode—it's one of the netlets named on its command line which it initialises when it starts.

It's useful to realise that while a bootstrap netlet is very important, it's is not part of the Jtrix definition. A bootstrap netlet is special to Jnode, which is a command line wrapper (plus more) around Nodality, but the Jtrix definition just defines what nodes and netlets etc should do. It doesn't specify that nodes have to have a command line wrapper any more than it specifies that anything related to Jtrix has to end with "ix", so it certainly doesn't specify what command line netlets ought to be able to do.

When a netlet is named on Jnode's command line it can be followed by a number of parameters of the form *name=value*, where each *value* is a I/O stream—in practice a file either for reading or writing. In our case we want to read in a warrant which the netlet will then use to access the hello service. It will look like this:

```
% jnode 203 -netlet-stdio hello3-client.xml warrant-in=hello-warrant.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting hello3-client
reading hello-warrant.xml for warrant-in: Warrant for service
Bootstrap complete
netlet:203.0.1: Hello, world, and thank you for using our service
^C

%
```

Bootstrap netlets are useful because netlets cannot choose to access system resources, and hence cannot read files. But whoever runs Jnode can choose to give certain netlets access to selected files, and this is how it's achieved.

In the example above the warrant comes in on an I/O stream, the netlet reads it, binds the service and gets the message. This is all very nice, but it raises some questions.

First, if netlets run in their own sandbox, why are these bootstrap netlets allowed to access these files?

Second, if netlets run in their own sandbox, what kind of mechanism allows them access to these files?

And third, if bootstrap netlets are particular to Jnode, how can we write a netlet which both conforms to Jtrix and still has access the necessary file access?

We'll answer these in order. . .

First, it's true that netlets run in their own protective sandbox. But bootstrap netlets are allowed this file access because they're being chosen to run by the node's owner (i.e. whoever executes Jnode) and if they don't want those netlets to run, then they shouldn't give them on the command line. Also—and more importantly—the node's owner chooses which files they are given, and they don't have any more file access than that. So access is still very tightly controlled.

Second, while netlets run in their own sandbox there is one exception. This is the external netlet which wraps the node, and that's only because it's a normal application as well as being seen as a netlet by the node. And Jnode is that external netlet. So it's Jnode which looks at the command line, interprets the *name=value* arguments, and can make them available to the bootstrap netlets.

And finally, how is this achieved? It uses netlet facets, those facets which a netlet offers a node, which the node can find with the `getFacets()` method of the `INetlet` interface and can bind with the `bindFacet()` method. By creating a netlet with the right netlet facet we are effectively offering that facet to any node we happen to be running on—and it still fits the `INetlet` interface we've seen before. If that netlet happens to execute in a different kind of node, or if it manages to execute in Jnode but not as a bootstrap netlet, then there's no problem—that netlet facet just doesn't get used.

8.1.2 Interfaces for bootstrapping and I/O

The facet which Jnode looks for is called `IBootstrapFacet`, in package `org.jtrix.facets1.netlet`.

Our netlet needs to implement and offer an `IBootstrapFacet` if it is to make use of being a bootstrap netlet and read in a warrant. This means it needs two methods. The `boot()` method is called by the node and allows us to do all the work. The `getInvocationTimeout()` method simply tells the node how long it can expect the `boot()` call to last before it can deem the netlet to have gone wrong.

When the netlet receives the boot call it's given an interface called `IManager`. This is an inner class of `IBootstrapFacet`, and just allows the netlet to ask the node for the relevant I/O streams. Looking at the run above we can see the warrant file *hello-warrant.xml* is given as an argument called *warrant-in*. So in our case, our netlet must ask for the *warrant-in* stream and it will be granted read access to that file. We could have called that stream anything, not just *warrant-in*, but whatever we call it both the person running Jnode and the netlet must agree on it.

So, the procedure so far: our netlet offers an `IBootstrapFacet`. The node binds it and calls its `boot()` method, giving us an `IManager` interface. Our netlet asks this `IManager` interface for the input stream called *warrant-in*, and thus can read the warrant.

Now, in an ideal world we would ask for this input stream and get an ordinary `java.io.InputStream` object from which to read the warrant. Sadly, this is not an ideal world. In particular, `java.io.InputStream` is not serializable and that means it can't pass through facets. And if it can't get through facets then it can't get into or out of a netlet. Similarly for `java.io.OutputStream`. What to do?

Luckily, Jtrix has a few tricks up its sleeve. First, it has a couple of facets called `IInputStream` and `IOutputStream`. These can be found in `org.jtrix.facets1.util.io`. This is what we get from `IManager`.

Second, it has utility classes `FacetInputStreamWrapper` and `FacetOutputStreamWrapper` in `org.jtrix.project.libjtrix.io`. These are subclasses of `java.io.InputStream` and `java.io.OutputStream` and are constructed by giving an `IInputStream` and `IOutputStream`. Thus if we're given an `IInputStream` we can use it to construct a `FacetInputStreamWrapper` and instantly we have an ordinary `java.io.InputStream`:

```
IInputStream facet1 = // Some input stream from a facet (and it is a facet)
InputStream stream1 = new FacetInputStreamWrapper(facet1);
// Now variable stream1 is an ordinary java.io.InputStream

IOutputStream facet2 = // An output stream from a facet (and it is a facet)
OutputStream stream2 = new FacetOutputStreamWrapper(facet2);
// Now variable stream2 is an ordinary java.io.OutputStream

// The Warrant object in org.jtrix.base takes a java.io.InputStream
// in its constructor. So if stream1 is reading a warrant XML file
// then we get a Warrant like this:

Warrant w = new Warrant(stream1);
```

The code below shows this in context.

8.1.3 About our code

We only need one public class, a client netlet we'll call `Hello3Client`, and it works like this:

1. Its `initialise()` method is minimal. It does (almost) nothing but return true.
2. Meanwhile, our netlet offers the facet `IBootstrapFacet`, which the node will discover if it checks our `getFacets()` method. And in facet when Jnode runs a bootstrap netlet that's exactly the facet it does look for.
3. Jnode will try to bind the facet by calling the `bindFacet()` method of our netlet. We have to return some object which implements `IBootstrapFacet`, so we'll return something called `WarrantReader`. This is an inner class. We also make sure we don't merely return it as-is—we wrap it in a `FacetHandle`, which we must always do when we return a facet to the world outside our netlet.
4. Since the `WarrantReader` implements `IBootstrapFacet` it has a `boot()` method which the node will call. It will pass in an `IManager` and from this we'll take the `IInputStream` called *warrant-in*.
5. We'll read the warrant through this input stream, bind it, and use it.

And finally a note on sequence: the node will always wait until the `initialise()` method is complete before it calls any other methods. So our `IBootstrapFacet` will be called after the `initialise()` method has returned. Actually, there is one exception to this: we might get terminated before the `initialise()` is done, but that's understandable, because termination could happen for any number of reasons and can't always be expected to be polite.

8.2 Hello3Client

The class starts with these imports. By now we have discussed all of these:

```
package org.jtrix.project.helloworld;

import org.jtrix.base.*;
import org.jtrix.facets1.util.io.IInputStream;
import org.jtrix.facets1.netlet.IBootstrapFacet;
import org.jtrix.project.libjtrix.io.FacetInputStreamWrapper;
import org.jtrix.project.libjtrix.netlet.NullService;

/** Bootstrap netlet which accesses a hello world service using a warrant
 * read from a bootstrap input stream. That stream is called "warrant-in".
 */
public class Hello3Client implements INetlet
{
    private INode _node;
```

Most of the `INetlet` methods are minimal. Even the `initialise()` method is tiny, although it does save the link to the node for reference later on:

```
    public byte[] initialise(INode node, Object bean, byte[] unsigned)
        throws InitialiseException
    {
        _node = node;
        return null;
    }

    public void terminate(long date, INetlet.IShutdownProgress progress)
    {
        // Nothing to clean up
    }

    public IService bindService(Warrant warrant, IService consumer)
        throws ServiceBindException
    {
        // No services for other netlets
        throw new ServiceBindException();
    }
```

It gets interesting around the other `INetlet` methods, because this is the first time we've offered a netlet facet. Our two methods tell the node what facet we offer, and deal with a request for it:

```
/** Lets a node see what facets we offer it.
 * @return An array naming the one facet we do offer the node: the facet
 *      through which we can access the I/O streams at bootstrap time.
 */
public String[] getFacets()
{
    return new String[]{ IBootstrapFacet.class.getName() };
}

/** Allows the node to bind any facets we can offer it.
 * @param facet  The name of the facet the node wishes to bind.
 * @return  An interface giving the node access to that facet.
 */
public IRemote bindFacet(String facet) throws FacetBindException
{
    if (facet.equals(IBootstrapFacet.class.getName()))
    {
        return new FacetHandle(new WarrantReader(), facet);
    }
    else
    {
        throw new FacetBindException();
    }
}
```

Notice, again, our use of `FacetHandle`. When the node binds our `IBootstrapFacet` we wrap the implenting object in a `FacetHandle` to help it through the mediation.

And here is the implementing class, the `WarrantReader`. The `boot()` method is called by the node which gives us an `IManager`, and from that we get the appropriate input stream containing the warrant:

```
/**
 * Read a warrant from an input stream granted to us by the node when it
 * boots.
 */
private class WarrantReader implements IBootstrapFacet
{
    /** The longest the node can expect the boot method call to last.
     * @return  Number of milliseconds after which if the boot method
     *      has not finished the node can stop it itself.
     */
    public long getBootInvocationTimeout()
    {
        return (5*60*1000);
    }

    /** Use the bootstrap system to pick up the input stream named
     * "warrant-in".
     * @param mgr  The object that allows us to access the I/O streams.
     * @return  Success flag.
     */
    public boolean boot(IBootstrapFacet.IManager mgr)
    {
        try
        {
            // Read the warrant from the "warrant-in" bootstrap parameter

            IInputStream is = mgr.getInputStream("Warrant for service",
                                                 "warrant-in");
            Warrant warrant = new Warrant(new FacetInputStreamWrapper(is));

            // Use the warrant to bind the service, and then a facet
```

```
                // we know is in the service

                IService service = _node.bindService(warrant, new NullService());
                String hf_name = IHelloFacet.class.getName();
                IHelloFacet facet = (IHelloFacet)(service.bindFacet(hf_name));

                // Use the facet
                System.out.println(facet.getMessage());
                return true;
            }
            catch (Exception e)
            {
                e.printStackTrace();
                return false;
            }
        } // boot()

    } // WarrantReader

} // Hello3Client
```

Notice that this use of the service is exactly the same as in our original `Hello1Client`. All that's different in this example is the way we got the warrant.

8.3 Compiling the application

For compilation to work we need these JARs on our classpath:

1. *jtrix.jar* which defines `INetlet` etc.
2. *facets1.jar* which contains `IBootstrapFacet` and `IInputStream`.
3. *libjtrix.jar* because we use its utility classes `FacetInputStream-Wrapper` and `NullService`. Recall we also used the `NullService` in `Hello1Client`, and for the same job.

We'll put our `Hello3Client` into *hello3.jar* along with `IHelloFacet`. As usual we don't need a manifest in this JAR.

8.4 Creating the netlet descriptor

Using *jtrixmaker* to create our netlet descriptor we need to tell it:

- What type of thing we're making. A netlet descriptor, as usual.
- What that descriptor will be called: *hello3-client.xml*.

- Where the relevant JARs are: */usr/lib/jtrix* (in our case) and the current directory.
- What JARs the netlet needs to run: *facet1.jar* for the bootstrap facets, *libjtrix.jar* for the `NullService` and of course *hello3.jar* for `Hello3Client`.
- The class name of our netlet, `Hello3Client`, fully qualified as always.

Now here's how we run *jtrixmaker*. Notice that have the *hello-warrant.xml* ready for when we run the netlet.

```
% ls
hello-warrant.xml  hello3.jar
% jtrixmaker -type netlet -outfile hello3-client.xml \
     -jardirs /usr/lib/jtrix . -jars facets1.jar libjtrix.jar hello3.jar \
     -classname org.jtrix.project.helloworld.Hello3Client
% ls
hello-warrant.xml  hello3-client.xml  hello3.jar
%
```

8.5 Running with Jnode

Execution looks like this:

```
% jnode 203 -netlet-stdio hello3-client.xml warrant-in=hello-warrant.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting hello3-client
reading hello-warrant.xml for warrant-in: Warrant for service
Bootstrap complete
netlet:203.0.1: Hello, world, and thank you for using our service
^C

%
```

Clearly what's new is feeding in the warrant on the command line, whereas with `Hello1Client` we had to build it into the netlet descriptor as the parameter bean. And as usual we terminate it with a Control-C.

8.6 Summary

If you've run the examples in *Start running Jtrix* then you will recognise this *name=value* approach on the Jnode command line. It's used by the hosting service, SAS and other applications that want to bootstrap themselves, providing a relatively easy way to get a netlet up and running with arbitrary configuration data.

Chapter 9

Debugging

Some tips on debugging netlets.

9.1 Standard I/O

The node runs in its own window and outputs any of its netlets' standard output (STDOUT) and standard error (STDERR). Output is prefixed by dotted identifier *a.b.c*. Each netlet has its own *a.b.c* combination: *a* is the node number, *b* is the accounting group within that, and *c* the netlet number within that contract. This is useful for watching debug information from various netlets simultaneously.

We also need to tell Jnode to display standard I/O for netlets by using the *–netlet-stdio* option. Have a look at Appendix A for this and other Jnode options.

9.2 Event messages

Of course, this is only useful if we're running a netlet on our own node, and we should really test an application thoroughly on our own nodes before sending them off elsewhere.

If we are using a remote node then it will save the last few STDIO messages for us. The console launcher application has a command, *dmesg*, to pick up these debug messages. Hence we can launch Jtrix applications from a local machine to a remote node and read off its remote debug information from the comfort of our local machine.

This is handy for debugging, logging, etc. However, the hosting service cannot reasonably hold all messages indefinitely, so we can expect it to drop messages if they are not picked up in time.

This facilility is actually a specialised version of a more general feature of the hosting service which enables various events to be gathered, then released via `IHostingControlFacet`. The *dmesg* command requests a `StdioEvent`, but others are defined in `org.jtrix.project.nodality.facet.audit`.

9.3 The Debug class

This is in package `org.jtrix.project.libjtrix.debug`. It is used to output messages at various "levels", and the developer can then select which levels they want to see when they run the application.

To turn debugging on fully and use it we use something like these lines:

```
// Enable debug output. The arguments are an identifier string, the bits
// to enable, and the bits to disable.  -1 as the first argument means
// enable all arguments. There are 63 useful bits (it's a long). At the
// time of writing these include INFO and WARN among others.

Debug.set("My netlet", -1, 0);

// The msg() method outputs a debug message at the appropriate level.
// The first argument is the level, the second is the object being
// debugged. Subsequent arguments are turned into strings and are output
// as the debug message. Here are two examples.

Debug.msg(Debug.INFO, this, "Inside initialise() with bean: ", _bean.toString());
Debug.msg(Debug.INFO, this, "Binding service...");

// Here's a special Debug method to handle debugging an exception.

try
{
    // Some code
}
catch (Exception e)
{
    Debug.exc(this, e, "Some code failed");
    // Handle exception
}
```

Only after first setting debugging will we get any debug output, and then it will appear in the node running the application. Without this only things explicitly printed to standard output will be seen.

We could also use the *dmesg* command as described above.

Beatrix provides some additional help with this class, including a command line option to set the debug level when we launch an application. See *Programming with Jtrix: The Beatrix application framework* for more details.

9.4 Miscellaneous tips

For general troubleshooting tips have a look at the *Troubleshooting* section in *Start running Jtrix: A practical guide*.

Additionally:

- Jnode caches JARs. It is intelligent about spotting differences between different JARs of the same name, but just to be safe you might want to delete its JAR cache directory if you've updated any JARs yourself. The cache directory is where your version of Java normally keeps its temporary files, in a subdirectory called *jtrix/cache*. You can remove this cache directory quite happily
- After recreating a JAR you also need to recreate any warrants and descriptors that refer to it. This is because the an old descriptor (or warrant) will refer to the old JAR's old hash code. If Jnode reads an old descriptor with an old hash code then it will try to use a previously cached version of that JAR. Thus your new JAR will not be used. This can be confusing.

Chapter 10

Threads

Threads are important in Jtrix because (a) we're working in a distributed environment, and (b) excessive thread use makes a netlet more expensive to run. Luckily Jtrix provides some rather clever tools to help manage threads. We split them into two camps, and we'll look at both in this chapter:

- Managing concurrency. Any netlet can be a server of some kind. For example, our `Hello2Provider` (Section 7.3) is a netlet providing a message. It could be invoked any number of times simultaneously by clients leading to excessive use. Fortunately, Jtrix lets us control this.
- Asynchronous facets. Normally, invoking a method on a facet will occur synchronously. That is, our netlet waits for the method to return and then continues. But we can also invoke a facet method asynchronously. That is, our netlet calls the method and returns immediately. Some time later the method call may return, and our netlet can handle it then.

10.1 Managing netlet concurrency

A netlet can manage its own concurrency using the method `setConcurrency(int threads, boolean thread_reuse)` in the `INode` interface. See Section 4.3.5.

10.1.1 Number of threads

The method allows the netlet to determine how many threads the node should start on its behalf. This is nothing to do with how many threads the netlet itself can start—it's completely free to start as many threads of its own as it wishes. But this `threads` parameter determines how many threads the node can start.

For example, our `Hello2Provider` offers a facet for others to use, the facet with the `getMessage()` method. When a client calls that `getMessage()` method the node starts another thread for `Hello2Provider` and executes that method. If two hundred clients call it simultaneously the node could start two hundred such threads, which may be very undesirable.

But our provider netlet can avoid this with the `setConcurrency()` method. If we set `threads` to 10 then the node will start no more than 10 threads for the provider. If there are 10 threads running and an 11th thread needs starting it will wait until one of the existing ones completes.

When we talk about the number of threads we include all threads started by the node for that netlet. This includes:

- The thread running `INetlet.initialise()`. When this method finishes, then that thread of course becomes free. This is the case even if it has started any other threads of its own.
- The same thread which executes the `setConcurrency()` method, or its parent, or parent's parent, etc, which was ultimately started by the node.
- Any incoming method invocation from a client netlet using us as a server.
- Any callback from which we invoked an outgoing asynchronous method call (see next section).
- A call from the node to our `INetlet.terminate()` to tell us our netlet is about to terminate. Of course, if this method cannot be invoked because there are no free threads then it doesn't stop us being terminated—it just stops us anticipating it.

Clearly, setting `threads` to zero or less doesn't make sense. Don't do it.

Note: At the time of writing the default concurrency level is 100, which is quite enough. If you're using more than 100 threads then it's probably time to rework your application.

10.1.2 Thread reuse

The `thread_reuse` parameter of `setConcurrency()` gives us further control. This boolean tells the node whether it can reuse a waiting thread (true) or if all threads should be started as new (false). The default is no thread reuse (false).

For example, suppose netlet A calls a facet method from netlet B which, in order to complete, needs to call back to A. Then in the normal state of affairs there are three threads:

1. Thread A1 is the initial calling thread. It runs in netlet A's processing space.
2. The method call starts a thread B1 which is entirely within B's processing space. Thread B1 does not count against netlet A at all.
3. Thread A2, which is called by thread B1 in netlet B, but which is a thread entirely within netlet A's space.

At this stage netlet A has two threads and B has one thread.

But this could be made more efficient if we were to enable thread reuse. Then when B1 calls netlet A it's the original thread, A1 which does the work. Thus A only uses one thread.

Nodality is quite smart about reusing threads. So in the above example the call sequence goes from A to B to A, then unwinds back to A when the last call is complete, and Nodality won't reuse a thread from a different calling sequence.

Thread reuse can also be helpful for avoiding deadlocks. Take the above A1–B1–A2 example with no reuse. Suppose thread A1 was the 10th thread and we had set a concurrency level of 10. Then B1 would wait for a free thread, which may never happen, thus creating a deadlock. If we had thread reuse then A1 could be reused and B1 could complete happily.

Thread reuse also preserves use of synchronize monitors. Take the situation A1–B1–A1 (thread reuse). If the first use of A1 held a synchronize monitor then the second use of A1 would still have that same monitor.

10.1.3 When should thread reuse be avoided?

By default threads are not reused. Thread reuse seems efficient, so why not make it the default?

To answer this, consider a situation similar to the one we've just seen:

1. A thread A1 is running in netlet A and calls into netlet B.
2. A new thread B1 starts in netlet B to respond to this. In its turn it calls into netlet A.
3. Netlet A is reusing threads, so the node resuses the thread A1 again.

So, A calls B and B which causes B to call A, while netlet A only has one thread running in it, thread A1, which is being reused for this latest call.

Now, here's something we've not mentioned: the node is so protective towards its netlets, that if A calls B and B takes too long to respond then the call will timeout—netlet A will get a `TimeoutException` (see package `org.jtrix.project.libjtrix.sync`). This is very nice of the node, because netlet A doesn't want to hang just because someone else's netlet is poorly written.

Back to our example: A calls B and B calls A, which is reusing its thread A1. Now suppose at this point thread A1 gets into a deadlock, or an infinite "while" loop, or for some other reason doesn't return. Then thread B1 will get a `TimeoutException` and, assuming B1 doesn't catch this, the same exception gets percolated back to A1, the original calling thread. But thread A1 is locked up, so it never receives that exception; it just carries on in its deadlock.

What's happened is that netlet A has been caught out by its own problematic code. The original call was from thread A1, thinking it was calling netlet B. But netlet B had to speak to netlet A which had a deadlock problem. Netlet B escaped from this deadlock thanks to the node's timeout protection, but netlet A was not so lucky.

The solution here is that A should not have been reusing threads. If A was not reusing threads then the call from B to A would have started a new thread, A2. Thread A2 would still have got deadlocked, but at least thread A1, being a different thread, would have found out about it.

This is why threads are not reused by default: because it's safer. Once you have thoroughly tested your code then you may choose to reuse threads.

10.1.4 Mediation timeouts

As mentioned above, a node protects a netlet from unreliable external code by throwing a `TimeoutException` if that external node takes too long. This is all part of the job of the mediator.

We won't go into details here. Suffice it to say that the timeout can be changed. At the time of writing the default is 30 seconds. This is part of the interface `IAsynchronous.IClient` which we cover next, albeit for other reasons.

10.2 Asynchronous facets

As we have seen so far, method invocation is normally synchronous (we have wait for the method to return), but Jtrix allows it to be asynchronous (not waiting for the method to return, and having to react only if and when it does later).

10.2.1 How to invoke asynchronously

So the procedure is: bind the facet, call the method, carry on. But how does Jtrix achieve this? What do we need to do? And how do we get to find out about the return?

It is all achieved with the magic of the mediator. Until now we have always talked about the mediator as being a boundary between netlets (and between the node and its netlets) and making sure they don't interfere with each other. And we've spoken about the `IRemote` proxy object that a mediator returns when we bind a facet, which we then cast into the type we want before using it.

But that mediated proxy can do much more than that. It acts as our interface into the other netlet and can help us invoke its facet methods asynchronously, among other things. When we perform an asynchronous invocation we supply a callback object which will get called whenever method returns.

Here's a code snippet. First it reminds us how to use a facet normally, then it shows us how to use it asynchronously. At this stage, it's very high level:

```
// How to use a facet normally: (1) bind it, (2) cast it into the right type.
// This should be very familiar to us.

IRemote proxy = node.bindFacet(IHelloFacet.class.getName());
IHelloFacet normal_version = (IHelloFacet)proxy;

// But we can take that same facet and make it usable asynchronously...

IAsynchronous.IClient async_version = (IAsynchronous.IClient)proxy;

// Now we can use invoke its methods asynchronously.  Variable "method"
// is the method we want to invoke. It's of class java.lang.reflect.Method.
// Variable "params" is an Object array. It contains the parameters we want
// to pass to the method. Variable "handler" is the callback object which
```

```
    // will get called when the method is invoked.

    async_version.invoke(method, params, handler);

    // And we carry on. Hopefully the handler object will get called eventually.
```

The first interesting thing is the cast into a `IAsynchronous.IClient`. It seems that the proxy object has a few hidden talents, and can be cast into more than just a normal facet interface. The `IAsynchronous` interface can be found in `org.jtrix.base`, and we're using its inner class/interface for clients.

The `invoke()` method of `IAsynchronous.IClient` simply lets us invoke the desired method in an asynchronous manner. If you've used `java.lang.reflect` before then you should be familiar with the broad principles of this. We say what method we're invoking along with its parameters.

The callback object will handle the method's return, when it happens. The node will start this in a new thread, or reuse one if appropriate.

Next, we present a complete netlet example which shows all the details in practice.

10.2.2 Asynchronous invocation example

This example is amazingly similar to the node facet example (`Hello2-Client` etc, Chapter 7) but we'll use a new facet for which threading is unavoidable.

Here is the facet interface we'll use:

```
package org.jtrix.project.documentation.threading;

import org.jtrix.base.*;

/** Facet interface for holding a secret.
 */
public interface ISecretHolderFacet extends IRemote
{
    /** Get the secret. However, this will block until unblock() is called.
     */
    public String getSecret();

    /** Free the secret. After this is called, getSecret() will return happily.
     */
    public void unblock();
}
```

The plan is to call `getSecret()` followed by `unblock()`. But clearly we have to work asynchronously, otherwise we'll never get to that second call. (Of course, if we call `unblock()` first then we can do without this kind of threading, but that would defeat the point of this example.)

Here's an ordinary Java implementation of the facet:

```
package org.jtrix.project.documentation.threading;

import org.jtrix.base.*;

/** Implementation of ISecretHolderFacet.
 */
public class SecretHolder implements ISecretHolderFacet
{
    private static final String _SECRET = "Send three and fourpence";
    private boolean _is_blocked = true;

    public synchronized String getSecret()
    {
        while (_is_blocked)
        {
            try
            {
                wait();
            }
            catch (InterruptedException e)
            {
                return "Interrupted while waiting for secret";
            }
        }

        return _SECRET;
    }

    public synchronized void unblock()
    {
        _is_blocked = false;
        notify();
    }
}
```

This is nothing special as far as threaded code goes.

Here's another very ordinary piece of code. It's just a netlet which makes the facet available as a node facet. It's almost identical to `Hello2Provider` from Chapter 7:

```
package org.jtrix.project.documentation.threading;

import org.jtrix.base.*;
import org.jtrix.project.nodality.facet.INodeAdminFacet;
import org.jtrix.project.nodality.facet.INodeFacetCollection;
import org.jtrix.project.nodality.facet.NetletID;

/** Offer a SecretHolder as a node facet.
 */
public class SecretProvider implements INetlet
{
    public byte[] initialise(INode node, Object bean, byte[] unsigned)
        throws InitialiseException
    {
        try
        {
            String na_name = INodeAdminFacet.class.getName();
            INodeAdminFacet na = (INodeAdminFacet)node.bindFacet(na_name);
            na.addNodeFacets(new FacetProvider(), null);
        }
        catch (Exception e)
        {
            System.out.println("Netlet failed to start");
            e.printStackTrace();
```

```
            throw new InitialiseException(e.toString());
        }

        System.out.println("Netlet started");
        return null;
    }

    public void terminate(long date, IShutdownProgress progress)
    {
        // Nothing to clean up when we terminate
    }

    public IService bindService(Warrant warrant, IService consumer)
        throws ServiceBindException
    {
        throw new ServiceBindException();
    }

    public String[] getFacets()
    {
        return new String[0];
    }

    public IRemote bindFacet(String facet) throws FacetBindException
    {
        throw new FacetBindException();
    }

    /** Provides a secret holder facet to the node.
     */
    private class FacetProvider implements INodeFacetCollection
    {
        private final String _facet_name = ISecretHolderFacet.class.getName();

        /** Say what facets we offer.
         */
        public String[] getFacets()
        {
            return new String[]{ _facet_name };
        }

        /** Bind the named node facet, as requested by a particular netlet.
         */
        public IRemote bindNodeFacet(NetletID id, String facet)
            throws FacetBindException
        {
            System.out.println("Instantiating "+facet);

            if(!facet.equals(_facet_name))
            {
                throw new FacetBindException();
            }

            System.out.println("Facet instantiated");

            return new FacetHandle(new SecretHolder(), facet);
        }

    } // FacetProvider

} // SecretProvider
```

So now the stage is set. We have a netlet which advertises a facet to the node, and we're going to have to invoke its methods asynchronously.

Here's the start of the client netlet. All the exciting work is deferred til

later, in method `demoAsync()`. All this does so far is bind the facet and provide the other aspects of a simple netlet:

```
package org.jtrix.project.documentation.threading;

import org.jtrix.base.*;
import java.lang.reflect.*;

/** Invoke a facet's methods asynchronously.
 */
public class SecretClient implements INetlet
{
    public byte[] initialise(INode node, Object bean, byte[] unsigned)
        throws InitialiseException
    {
        try
        {
            String facet_name = ISecretHolderFacet.class.getName();
            IRemote proxy = (IRemote)(node.bindFacet(facet_name));
            demoAsyncClient(proxy);
        }
        catch(Exception e)
        {
            System.out.println("Sorry, couldn't demonstrate async clients.");
            e.printStackTrace();
            throw new InitialiseException(e.toString());
        }

        return null;
    }

    public void terminate(long date, IShutdownProgress progress)
    {
        // No clean-up necessary
    }

    public IService bindService(Warrant warrant, IService consumer)
        throws ServiceBindException
    {
        throw new ServiceBindException();
    }

    public String[] getFacets()
    {
        return new String[0];
    }

    public IRemote bindFacet(String facet) throws FacetBindException
    {
        throw new FacetBindException();
    }
```

Here's the guts of it. Let's have a look first and discuss it after:

```
    /** Demonstrate how to take a mediated proxy and invoke it asynchronously.
     * @param proxy  Any mediated proxy, such as return by a bindFacet() call.
     */
    private void demoAsyncClient(IRemote proxy) throws Exception
    {
        // Turn the proxy into an asynchronous client and call it asynchronously

        IAsynchronous.IClient async_facet = (IAsynchronous.IClient)proxy;
        Method meth = ISecretHolderFacet.class.getMethod("getSecret", new Class[0]);
        async_facet.invoke(meth, new Class[0], new ReturnHandler());
```

```
        // Now take the same proxy and call it normally.

        ISecretHolderFacet normal_facet = (ISecretHolderFacet)proxy;
        normal_facet.unblock();
    }
```

The method takes the mediated proxy as a parameter. That is, it deals with the result of a `bindFacet()` call, which is the proxy given to us by the mediator. Normally we just cast this `IRemote` object straight into the type for the desired facet (such as `IHelloFacet`, `ISecretHolderFacet` and so on). But here we do something else first.

We can see there are two parts to the method: using the facet asynchronously, and using it normally. Let's focus on the asynchronous use first.

The cast on the first line gives us an asynchronous client, so we can invoke the `getSecret()` method appropriately. Next we get a `java.lang.reflect.Method` which simply identifies that method. Then we invoke it. We tell our asynchronous client the method we want to invoke, what parameters we're sending to the method (none) and a handler to deal with the result. We'll deal with the handler shortly.

So what we've done is invoke the `getSecret()` method asynchronously. That method won't return yet because we haven't called `unblock()`, but we carry on our execution regardless.

We're now in the second part. Here we simply cast the `IRemote` proxy into the appropriate class and unblock the secret. That's all we need to do.

Now here's the handler to deal with the return value of `getSecret()`:

```
    private class ReturnHandler implements IAsynchronous.IListener
    {
        /** Handle a successful return from the method invocation.
         */
        public void completed(Object rtn_value)
        {
            System.out.println("The secret is: "+(String)rtn_value);
        }

        /** Handle an exception or other throwable from the method invocation.
         */
        public void failed(Throwable t)
        {
            System.out.println("Sorry, couldn't get the secret.");
            t.printStackTrace();
        }
    }

} // SecretClient
```

We can see that this class needs to be an `IAsynchronous.IListener` to handle the return from an asynchronous call. We implement its two methods.

The `completed()` method handles a successful result—the method call returns okay. We handle it by simply taking the result value and outputting it. Since `getSecret()` returns a `String` we know we can safely cast it into a `String` and output it.

On the other hand our asynchronous call might have thrown an exception. This is passed to `failure()` which in our case simply reports the problem.

Note that this handler will of course start in a new thread. The remote `getSecret()` will return in the facet's implementation (in `SecretHolder`) and its return value gets passed back to us via the mediator and the node starts the thread in which `completed()` is run. And that also means (looking back at the last section on managing netlet concurrency) that we need to have our concurrency level set high enough, otherwise the node will wait until there is a thread free. Ordinarily this is not an issue—if we don't explicitly set our concurrency level then the node is happy to start all the threads it needs. But if we have previously called `setConcurrency()` then we need to be aware of this.

And that's the end of the class. To recap, we have a client netlet which binds a facet and invokes two methods, one of them asynchronously. It has a handler deals with the result of the asynchronous method call.

10.2.3 Preparing the example

To try this out, we first need to compile our classes and interface. We'll put all the above in a JAR file named *secret.jar*. Nothing else goes into this JAR.

Now we can use this to create two netlet descriptors: one for the provider and one for the client:

```
% ls
secret.jar
% jtrixmaker -type netlet -outfile secret-client.xml \
     -jardirs . -jars secret.jar \
     -classname org.jtrix.project.documentation.threading.SecretClient
% jtrixmaker -type netlet -outfile secret-provider.xml \
     -jardirs /usr/lib/jtrix . -jars nodality_facet.jar secret.jar \
     -classname org.jtrix.project.documentation.threading.SecretProvider
% ls
secret-client.xml  secret-provider.xml secret.jar
%
```

We'll run these on the same node.

10.2.4 Running the example

We run these examples as usual, starting a node with both netlets on it. Just as with the `Hello2Client` and `Hello2Provider` we run the provider first, so it can advertise the facet, then the client:

```
% jnode 206 -netlet-stdio secret-provider.xml secret-client.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting secret-provider
starting secret-client
netlet:206.0.1: Netlet started
netlet:206.0.1: Instantiating org.jtrix.project.documentation.threading.ISecretHolderFacet
netlet:206.0.1: Facet instantiated
Bootstrap complete
netlet:206.0.2: The secret is: Send three and fourpence
^C

%
```

We can see we have obtained the secret successfully. We terminate the node by hitting Control-C as usual.

Recall that output from the node and each netlet does not appear in sequence with each other, though they do appear in sequence within themselves. So nelet 1, the provider, will show “Netlet started”, “Instantiating...” and “Facet instantiated” always in that sequence. But the its message “Netlet started” may appear after the node has announced it’s starting the client netlet. Despite this, the node will only start netlet 2, the client, after netlet 1 has completed returned from its `initialise()` method.

10.3 What more can we do?

The mediator holds more exciting possibilities than those we’ve covered here.

The interface `IAsynchronous` holds the interface `IAsynchronous.IServer`. When the provider returns this (wrapped in a `FacetHandle` as always) it allows the server side to handle the method invocation asynchronously. It can then explicitly call an `IAsynchronous.IListener` method which is picked up by the client.

Also, thread reuse can be managed much in a much more fine-grained way than just setting a true/false flag.

We won’t cover either of these things here, but have a look at the Javadoc for details. Everything can be found in `org.jtrix.base`.

Chapter 11

System resources

Use of resources—the file system and networking—needs to be managed securely and needs to be accounted for. A node might want to allow netlets file system access, for example, but not use its local file system. By using particular patterns a Jtrix node can achieve this with minimal confusion for the netlets.

Here we look at how this is achieved with Nodality, plus how to embed a non-Jtrix application into a Jtrix environment.

All this is specific to Nodality. Some Jtrix applications will not run directly on Nodality; they will run on a hosting service which may or may not run on Nodality. In those cases the hosting service will give you resource access and manage them for you. The details here are lower level.

As a final aside, most serious applications are expected to use the resource management features of a hosting service, and if we build an application with the Beatrix framework, which uses that, then this becomes even easier. In these cases our application will run with any hosting service that implements the right interfaces. Node implementation becomes irrelevant.

11.1 Networking

Access to mediated networking is achieved with just this code:

```
import org.jtrix.facets1.node.ITrampolineFacet;
import org.jtrix.facets1.node.ISocketFactory;

...
```

```
String tf_name = ITrampolineFacet.class.getName();
ITrampolineFacet tf = (ITrampolineFacet) node.bindFacet(tf_name);
ISocketFactory sf = tf.getDefaultSocketFactory();
tf.setSocketFactory(sf);
```

That's it. Networking is initialised. After this the netlet's code can happily use the Java networking classes as usual.

The `ITrampolineFacet` accesses the trampolining mechanism for that netlet, allowing requests to be proxied (trampolined) to the right resources. The first two lines (after the import) simply bind to the netlet's trampoline system.

The second line then tells the facet to use its own socket factory for datagram and stream sockets.

One of the pleasing things about this approach is that the implementation behind the resulting trampoline facet is specific to that netlet. Another netlet running on the same node will get a different implementation when it also binds and sets. On the other hand, one netlet can still pass its received implementation to another netlet if it wishes.

When dealing with a server socket (not an ordinary socket) Java assumes its implementation does not change throughout the server socket's lifetime. When a server socket is created it waits for a connection; when a connection is made a new socket is created, and if the implementaion has changed in between then a `SocketException` is thrown, saying "Socket Implementaion Differs". This could be a problem if you change the socket factory. If you do need to change it then make sure your server sockets are closed before changing.

It is easy to imagine also a particular socket factory which manages use of two services simultaneously distinguishing them by their own port ranges, for example.

11.2 File system access

This is achieved with a few more lines than the networking example:

```
import org.jtrix.facets1.node.ITrampolineFacet;
import org.jtrix.project.nodality.facet.INodeResourceFacet;
import org.jtrix.facets1.util.io.IFileSystem;

...

String tf_name = ITrampolineFacet.class.getName();
ITrampolineFacet tf = (ITrampolineFacet) node.bindFacet(tf_name);

String rf_name = INodeResourceFacet.class.getName();
```

```
INodeResourceFacet rf = (INodeResourceFacet) node.bindFacet(rf_name);

INodeResourceFacet.IAccountedFileSystem
 tfs = rf.createFileSystem("/tmp", false);

tf.mountFileSystem("/myhome",tfs.getFileSystem());

...

tf.unmountFileSystem("/myhome");
```

After this, we can use all the usual Java file classes, such as `java.io.File` and `java.io.FileInputStream`, etc.

Following the imports, there are four steps in the code above:

1. Bind to the node's trampoline facet. This is the same as we did with networking.
2. Bind to Nodality's resource facet. This allows us to create resources.
3. Create our own file system on the node. In this example we want it to live in the node's */tmp* directory. We set the read-only flag to false.
4. Mount the file system. In this example we mount it onto the prefix */myhome*. Now we can write a file called, say, */myhome/profile.txt*. It will really be written to the node's */tmp/profile.txt*. All our file reading and writing is done with the usual Java classes and methods.

When we're done we can unmount the file system.

Some points to note in the above:

- Node directories

  Different systems are laid out differently. In the example above we got access to */tmp*. But if we were running on another system we might have had to ask for *C:\Temp* instead.

- Prefix name

  We chose our prefix to be */myhome* and therefore had access to */myhome/profile.txt*. If we had missed out the leading / then Jtrix would have added it for us. By contrast, only the leading character can be a /. Anything else will throw an exception. Similarly, the prefix may not contain a colon anywhere in it.

- Filenames

  Once we've mounted our file system Jtrix insists we access it as if it were a Unix file system, using forward-slashes. This is regardless of what the underlying system is.

- Several mount points

  Of course, several mount points can exist simultaneously, and these can be based in different parts of the host system. However, they cannot be mounted on top of each other—which is actually a corollary of the last point.

- */proc*

  The */proc* hierarchy is given to us by default; we don't have to bind or mount anything to access it. It contains process-specific files which at the time of writing are: (a) the subdirectory *runtime* which contains the Java runtime JAR, *rt.jar*; (b) the subdirectory *codebase* which contains all other JARs currently in use by this netlet, each named with the label that was used to download it.

- Nodality specific

  Steps 2 and 3 above are specific to Nodality. It is a Nodality feature to offer the option of local file system access (in this case */tmp*). If this code is running on another kind of node then those two steps wouldn't work. And furthermore, the node has. . .

- Right of refusal

  Access to */tmp* or *C:\Temp* or whatever else we tried to use is at the complete discretion of the node. It has every right to refuse us. And that's why we have. . .

- Freedom to switch providers

  Step 4 above mounts a file system, but that second parameter to `mountFileSystem()` doesn't have to be a Nodality file system. It can be anything which implements the facet `org.jtrix.facet.node.IFileSystem`. So a netlet might well start up with a warrant for a file system service which provides an `IFileSystem`. If this is in variable *myfs* then the code is reduced to this:

```
String tf_name = ITrampolineFacet.class.getName();
ITrampolineFacet tf = (ITrampolineFacet) node.bindFacet(tf_name);
tf.mountFileSystem("/myhome", myfs);

...

tf.unmountFileSystem("/myhome");
```

- Different providers simultaneously

  Several different providers can be used simultaneously on different mount points.

11.3 A note on resources

The resource-access methods here are specific to Nodality. They are useful if your netlet/application can rely on running on Nodality; Jnode is one such application.

But in practice it is envisaged that anyone could write a node, and netlets maybe running on any of these. A node-independent way of accessing system resources will be discussed in a future version of this document. In short the procedure is this: the node administrator creates or allocates some resources (disk space, IP addresses, etc) and then grants them to netlets as part of their hosting contract. To make an application more managable it is likely that all the resources will be taken by an application manger netlet which then controls the execution of its worker netlets, including distribution of resources to them.

11.4 Embedding a non-Jtrix application

Embedding an external application into Jtrix is simply a case of allocating resources (networking and disk) and then executing the `main()` method of the application. Here's an example.

11.4.1 The code

First, here's a simple non-Jtrix application:

```
package org.jtrix.project.documentation.embeddedapp;

import java.net.*;
import java.io.*;
import java.util.Date;

/**
 * Class that uses ordinary system resources, so we can try to
 * embed it into Jtrix.
 */

public class ResourceUser
{
    private static String _WEB_PAGE = "http://www.jtrix.org/index.htm";

    public static void main(String[] args)
    {
        try
        {
            InputStream is = (new URL(_WEB_PAGE)).openStream();
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            FileWriter file = new FileWriter("/myhome/output."
                                             +(new Date()).getTime()+".html");
            String line;
```

```
            while ((line = br.readLine()) != null)
            {
                file.write(line+"\n");
            }
            file.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

    } // main(String[])

} // ResourceUser
```

It is designed deliberately to use networking and disk space. Note that it also expects to have a directory called */myhome*. It could expect anything. We just need to make sure that when we map some disk space onto something called */myhome* for our application.

Here is the netlet which embeds the `ResourceUser`:

```
package org.jtrix.project.documentation.embeddedapp;

import org.jtrix.base.*;
import org.jtrix.facets1.node.ITrampolineFacet;
import org.jtrix.facets1.node.ISocketFactory;
import org.jtrix.project.nodality.facet.INodeResourceFacet;
import org.jtrix.facets1.util.io.IFileSystem;

/**
 * A netlet which demonstrates use of an external application.
 * Note this is specific to Nodality. Ordinarily an application
 * would expect to have resources provided by its hosting service.
 */

public class EmbeddedAppNetlet implements INetlet
{
    /** Initialise and execute the program. First we get a socket
     * factory and a filesystem which the program is going to use.
     * Then we can just execute the program. */

    public byte[] initialise (INode node, Object bean, byte[] untrusted)
        throws InitialiseException
    {
        try
        {
            String tf_name = ITrampolineFacet.class.getName();
            String fs_name = IFileSystem.class.getName();
            String rf_name = INodeResourceFacet.class.getName();

            // Use sockets

            ITrampolineFacet tf = (ITrampolineFacet) node.bindFacet(tf_name);
            ISocketFactory sf = tf.getDefaultSocketFactory();
            tf.setSocketFactory(sf);

            // Use files. Read-only = false

            INodeResourceFacet rf
                = (INodeResourceFacet) node.bindFacet(rf_name);
            INodeResourceFacet.IAccountedFileSystem tfs
                = rf.createFileSystem("/tmp", false);
            tf.mountFileSystem("/myhome",tfs.getFileSystem());
```

```
            // Run external program

            ResourceUser.main(new String[]{});

            tf.unmountFileSystem("/myhome");
            return null;
        }
        catch (Throwable e)
        {
            e.printStackTrace();
            throw new InitialiseException("Couldn't run netlet");
        }

    } // initialise

    /** Terminate; no actions required. We're 100% done as soon as we enter! */

    public void terminate(long expiry, INetlet.IShutdownProgress progress)
    {
        progress.performed(100);
    }

    /** We offer any service connections. */

    public IService bindService(Warrant w, IService s)
        throws ServiceBindException
    {
        throw new ServiceBindException("No services supported");
    }

    /** We offer no facets. */

    public String[] getFacets()
    {
        return new String[]{ };
    }

    public IRemote bindFacet(String facet_name)
        throws FacetBindException
    {
        throw new FacetBindException("Don't support facet "+facet_name);
    }

} // EmbeddedAppNetlet
```

It simply assigns some resources, making sure the disk space maps to */myhome*.

11.4.2 Compiling the code

Compiling this requires the following JARs on our class path:

- *jtrix.jar* as usual, because we use `INetlet`, etc.
- *nodality_facet.jar* because we're using Nodality facets.
- *facets1.jar* which contains the trampolining facets.

The non-Jtrix application doesn't need any special JARs, so that's all we need. We'll bundle it up into a JAR called *embedded_app.jar*.

11.4.3 Creating the descriptor

The netlet descriptor is created as follows. This also shows what files we do and don't have at this stage:

```
% ls
embedded_app.jar
% jtrixmaker -type netlet -outfile embedded-app.xml \
   -classname org.jtrix.project.documentation.embeddedapp.EmbeddedAppNetlet \
   -jardirs /usr/lib/jtrix . \
   -jars embedded_app.jar nodality_facet.jar facets1.jar
% ls
embedded-app.xml  embedded_app.jar
% ls /tmp/output.*
ls: /tmp/output.*: No such file or directory
%
```

Clearly there's no output file, yet; that's what the application is going to generate.

11.4.4 Running the application

So here's how we run the original program in its Jtrix environment:

```
% jnode 205 -netlet-stdio embedded-app.xml
Jnode starting...
Initialising Nodality...
Bootstrap starting
starting embedded-app
Bootstrap complete
^C

% ls /tmp/output.*
/tmp/output.1012310700417.html
%
```

As we can see, after we run the netlet with Jnode, and our ordinary application has indeed captured the Web page and written it to a file.

11.5 Summary

The embedded application is clearly contrived, but it does prove a point. Jtrix.org has put a more realistic application—the Tomcat servlet engine from The Jakarta Project—into Jtrix, thus making a mobile and redundant servlet engine. For more particular resource usage the reader is very strongly recommended to use the Beatrix application framework which handles this through a hosting service, plus redundancy, service access, warrant generation, administration interfaces, console access and more.

Appendix A

jnode_help.txt

Here is the help file for Jnode. For more practical details on these see *Start running Jtrix: A practical guide* at `http://www.jtrix.org`.

Appendix B

jtrixmaker_help.txt