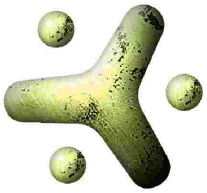


Programming with Jtrix:
The Beatrix application framework

January 2002, for release 0.14

$Id: jtrix-beatrix-programming.lyx,v 1.15 2002/02/08 16:07:21 nik Exp $

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

Contents

Chapter 1

Introduction

This document describes how to write Jtrix applications and services using the Beatrix framework.

A basic familiarity with Jtrix programming is assumed. Otherwise, here are some other documents you might like to refer to, all available from `http://www.jtrix.org`:

- *How to write netlets*—introduction to Jtrix and writing your first netlets. A pre-requisite for reading this.
- *Start running Jtrix*—how to run a node, Jtrix demo applications, etc and other practical information.
- *Jtrix: A technical overview*—technical motivation and concepts behind Jtrix.
- *Platform Specification*—formal description of what defines the base platform. Several key services are not described in this document, as they are additional to the platform.
- *The Jtrix Dictionary*—a handy guide to the terms used.

1.1 How to read this document

How this document is broken down, to help you skip the unwanted bits:

- Chapter 1: Introduction and orientation.
- Chapter 2: High level concepts, essential for understanding the principles of Beatrix.
- Chapter 3: An introduction to plugins, the basic building blocks of Beatrix applications.

- Chapter 4: A simple Beatrix application. Lots of new concepts here, and it won't be 100% understandable straight away, but this example application will drive us through the rest of the document.
- Chapter 5: Key plugins and interfaces. Essentially the Beatrix API, so this is really the heart of this document.
- Chapter 6: The launcher, which is a key admin interface to Beatrix applications.
- Chapter 7: Additional application aids and essential tools for working with Beatrix.

Have a look at the appendices, too.

1.2 What is Beatrix?

Writing applications is more than writing netlets: it's about finding hosting resources, keeping the right number of worker netlets running together with the right resources in the right locations, coping with outages, inter-netlet communication and so on.

The Beatrix framework is designed to make this easy. It provides a framework for a common pattern for distributed applications. It does this through a plugin architecture:

- All functionality is achieved via plugins.
- Many useful plugins are provided as standard.
- Plugins can be added dynamically to adapt functionality on the fly.
- Plugins can be added and managed remotely from one netlet to another.
- New plugins can of course be written very easily.

1.3 What is Beatrix good for?

Beatrix applications consist of a number of worker netlets kept in check by a number of manager netlets. One manager is a leader, but this leader doesn't have to be constant—it can always be replaced by another manager. Additionally access point netlets provide a link from third party consumer netlets who want to access our service. See Figure 1.1.

This pattern fits many kinds of application. Meanwhile several Beatrix applications could be linked up, so the options multiply even further.

If your application doesn't fit this pattern then you can write Jtrix netlets without Beatrix, or devise an alternative framework.

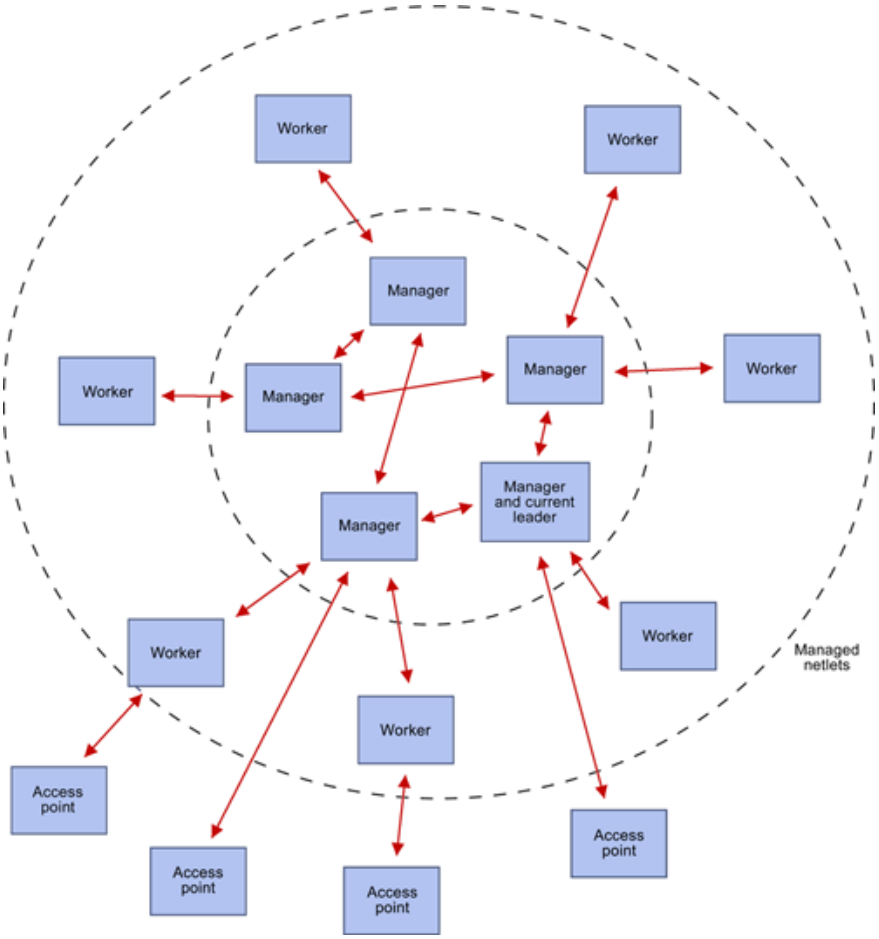


Figure 1.1: Relationship of netlets in a Beatrix application.

Chapter 2

High level concepts

Some basic concepts to help us understand the sample code.

2.1 Beatrix netlets

The Beatrix framework is for applications with the following pattern (Figure 1.1):

Access points download to client applications and act as proxies to the service-proper. Since they are so simple and are deployed transparently they can usually be ignored. However, access points may be specialised—see Section 7.4.

Workers do the bulk of the work and use system resources. Any one application is likely to have several different kinds of worker, each with a different job.

Managers manage the deployment of worker netlets. Their sole function is to keep them alive and healthy. Managers work in a group of one *leader* and several backups. Leadership elections occur transparently when the current leader leaves. There is only one kind of manager.

2.1.1 Netlet classification

Each of the above—access point, worker, manager—is a different *netlet type*.

Each different kind of worker or access point is a particular *worker type* or *access point type*. The phrase *netlet sub-type* is synonymous with worker type or access point type.

Each netlet type describes a *role*. Both workers and managers are *managed roles* because they are under the control of the Beatrix framework,

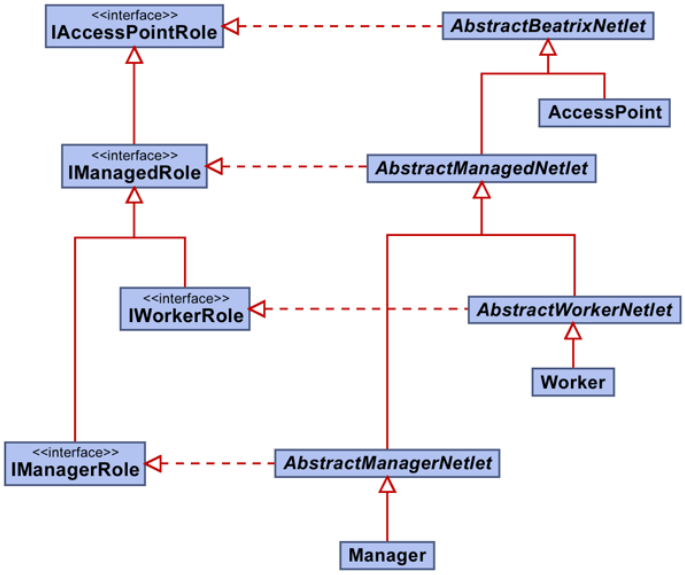


Figure 2.1: The Manager, Worker and AccessPoint classes, and how they relate.

whereas access points are created in response to external netlets requesting services.

Don't get confused between managed netlets and manager netlets—there's only a one letter difference.

See Figure 2.1 on how the netlet classes relate, although the vast majority of these are unlikely to be used directly in most applications.

2.1.2 Writing managed netlets

A Beatrix application is written in terms of *plugins*. Writing a worker or manager netlet involves adding plugins to basic existing netlets, whether they are our own plugins or off-the-shelf ones.

Communication is biased to allow worker–manager communication and manager–manager communication.

2.1.3 Access points

An access point netlet is created in response to a service bind from a consumer of our service or application, and downloaded to fulfil the required service.

In Beatrix they're just very simple proxies by default. Both workers and managers can act as *servers* for access points. If one server fails the access point re-establishes communications with another one.

Access points can, however, be custom-built, can use resources provided by the client netlets, and so on. For more on this see Section 7.4.

2.1.4 Workers

On start-up they are given system resources and told which of these, if any, are left from a previous worker in the same application, and which are new.

For example, a worker netlet may store data on a local file system. If it dies and is restarted the data may still be there from the last time and it can carry on. But it may be starting with a completely fresh system and therefore need to get the latest dataset from elsewhere before it starts work.

Beatrix provides a `Worker` class which has to have plugins added before it becomes useful.

2.1.5 Managers

Managers ensure there are enough workers doing the right jobs, and may also provide services. They do not use system resources, and instead leave that to the workers.

By default managers can communicate with the leader and the workers, but only the leader can communicate with other managers. Arbitrary combinations are possible, however.

Managers have three start-up modes:

Cold start When an application is being started for the first time. It needs to deploy backup managers and worker netlets, and generally get everything off the ground. This manager becomes the leader.

Joining When the service is already running. This manager needs to join the existing management group as a backup.

Warm start When the service was running until all the managers died. In this mode the manager has to revive the situation by getting the last service state, updating workers, and generally get everything back on track. Warm start is triggered not by an administrator, but by one of the hosting services—it notices application failure and then warm starts it.

Beatrix offers a basic `Manager` class which needs plugins added before it becomes usable.

2.2 Command line control

Beatrix provides a command line console through which an application can be launched and managed. We call this the *launcher* or *console*.

To launch an application for the first time it uses a *launch descriptor* which tells it what command line options are accepted at launch, plus details of all the netlets (their JARs, classes, roles) and services (interfaces, access points, etc) needed. See Section 4.9.

The console launcher also needs warrants for a SAS (service advertisement service) through which the service can be propagated, and a hosting service in which it will initially live. However, the SAS warrant can be omitted in certain circumstances.

As with all aspects of Jtrix connectivity is not assumed, and the console may get disconnected and need to be reconnected at another time and place. To allow this, a successful launch returns an admin warrant. Use of this allows console reconnection to the application at any time.

After launch the console can be used to manage the service. The console has commands of its own, and application-specific commands are handled by the application's managers and/or workers.

See Chapter 6 for more on the console.

2.3 Plugins

Beatrix uses plugins to simplify service sessions and resource usage. The plugin architecture starts with a ready-made Beatrix application into which plugins can be added for customised functionality.

A *plugin* is anything that implements the IPlugin interface.

A *plugin manager* is a holder into which the plugins are added. A plugin manager is part of a fully implemented Beatrix `Manager` netlet which runs an application. Whenever it needs to do a particular task such as deploy workers, deal with services, start up, etc, then it looks for an appropriate plugin (or plugins) in its plugin manager.

2.4 Anatomy of a Beatrix application

Figure 2.2 shows how our example application, "skeleton", is constructed.

This example application is detailed in Chapter 4, but in brief it provides a simple message consisting of the words "world" and "hello" among others.

The `Manager` and `Worker` classes are the standard Beatrix manager and worker netlets. Everything else is a plugin. Unmarked plugins

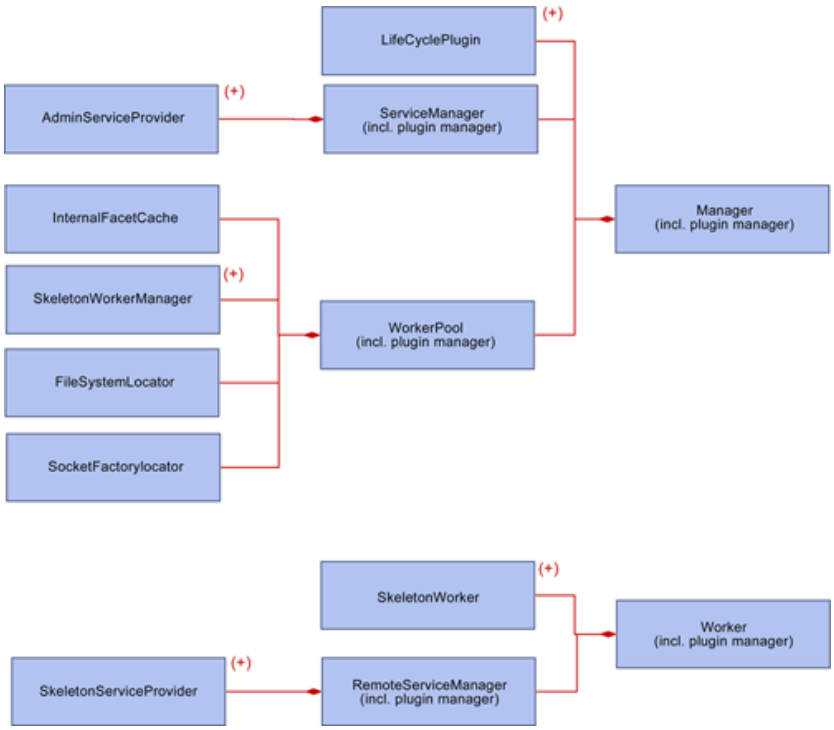


Figure 2.2: How the sample skeleton application is designed. Plugins marked (+) are those we have had to provide ourselves. The others are provided by Beatrix.

are provided with Beatrix, while those marked (+) we have written ourselves.

The `Manager` netlet's plugins are as follows:

LifeCyclePlugin (+) Starts up the application and adds all the other plugins. Flow control begins here. When start-up is complete all other flow is event-driven, perhaps by timed events (e.g. periodic checks on workers) or external events (e.g. a third party binding a service).

ServiceManager A collection point for all the services the manager will provide. In this case, only the `AdminServiceProvider`.

AdminServiceProvider (+) Allows the application's administrator to control it remotely from a console.

WorkerPool Each worker pool is a collection point for plugins, all of which aid in the management of a particular worker type. In this case we have only one worker type and hence only one worker pool plugin.

InternalFacetCache Beatrix allows netlets in the same application to communicate easily via facets internal to that application. The `InternalFacetCache` provides an extra tool to help this. It is plugged into the worker pool, which means it is accessible to all the other plugins in the worker pool.

SkeletonWorkerManager (+) The main plugin which manages the deployment and life cycle of our worker netlets. It responds to a worker start event by adding the worker's plugins remotely. Hence the worker turns from a dummy netlet into a useful netlet dynamically. This plugin ignores worker die events. It also determines resource requirements when new workers need to be started.

FileSystemLocator Helps the `SkeletonWorkerManager` find file system resources for workers. In fact, the way it is used in this example application means this plugin does all the work for the `SkeletonWorkerManager`, which therefore does nothing except configure it once.

SocketFactoryLocator Helps the the `SkeletonWorkerManager` find a socket factory resource for workers. This plugin can take the entire workload off the `SkeletonWorkerManager`, just like the `FileSystemLocator`. But for the sake of example we have not done this. Instead, the `SkeletonWorkerManager` uses it explicitly to locate socket factory resources, then assesses the results itself and decides whether they are suitable.

As descibed above, the `Worker` netlet is configured dynamically by the `SkeletonWorkerManager` which adds these plugins remotely:

SkeletonWorker (+) The first worker plugin to be started, and it is started remotely by the `SkeletonWorkerManager`. It binds the resources granted to it, but, since this is a dummy application, doesn't use them.

ServiceManager The same plugin as above, this time added to the `Worker` netlet.

SkeletonServiceProvider (+) Provides the main consumer service which in this case provides two facets. One is for applications to get the message; the other is for a user to get the message via a console command.

Chapter 3

An introduction to plugins

Writing a Beatrix application isn't usually more than writing a number of plugins. In this chapter we look at what they are and how to work with them.

3.1 What is a plugin?

A *plugin* is any class which implements the `IPlugin` interface in `org.jtrix.project.beatrix.plugins`. At this stage we don't need to worry about what's in this interface, although it's quite simple.

Plugins are collected together into a *plugin manager* object. Any object which implements `IPluginManager` (in the same package as `IPlugin`) is considered a plugin manager.

Any class which thinks it might benefit from having a plugin-style architecture can be a plugin manager, and any class which wants to provide "pluggable" features can be a plugin. Here are some examples:

- The class `WorkerPool` is a tool that organises groups of worker netlets. But there are some application-specific things it doesn't know: what kind of resources they need, if any extra action is needed when a new one starts, etc. So `WorkerPool` is a plugin manager. It relies on plugins to provide this extra intelligence.
- The class `ServiceManager` is a tool that handles service requests from consumers, checking security, etc. But it doesn't carry the functionality of the services themselves. Therefore it is a plugin manager, relying on plugins themselves to provide the service functionality.
- The class `Manager` represents a manager netlet. It knows it should provide workers and services, but they could be of various kinds. Therefore it is a plugin manager. It relies on worker pools and

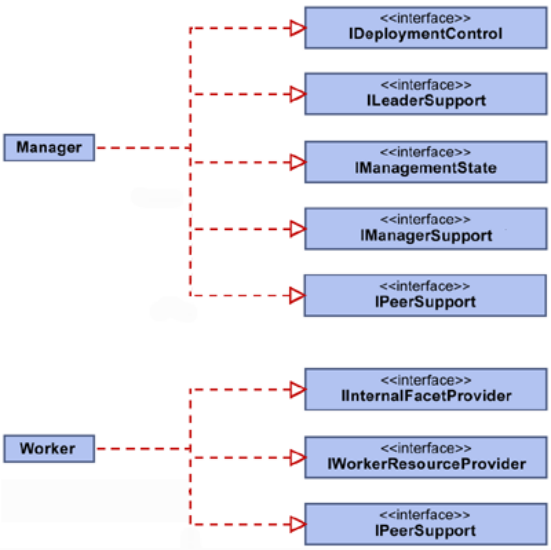


Figure 3.1: Plugins provided (and implemented) by the `Manager` *and* `Worker` *classes.*

service managers as plugins for those extra details. So we can see that `WorkerPool` and `ServiceManager` are plugins themselves as well as plugin managers.

3.2 The Manager and Worker classes

The `Manager` class is a Beatrix manager which enables the plugin architecture. It has its own plugin manager into which we can add our plugins. It also comes with a number of plugins already added.

The `Worker` class is the corresponding implementation of a worker netlet, and has a similar plugin design. It has a plugin manager with several plugins already there.

We can see in Figure 3.1 what plugins are provided by the `Manager` and `Worker`. That figure also shows us that those plugins are actually implemented by the `Manager` and `Worker`.

3.3 Flow control

Application initialisation begins inside the `Manager` class which, after some work, passes control to a plugin. When this finishes, initialisation is virtually complete.

Which plugin does it pass control to? Whichever plugin implements `IManagerLifeCycle`. If the application is cold-starting then its `coldStart()` method is called. If the application is warm-starting its `warmStart()` method is called.

We can see from Figure 3.1 that no `IManagerLifeCycle` plugin is supplied with the `Manager` netlet. This means we have to implement it ourselves and make sure we add it into the `Manager` netlet.

After the `IManagerLifeCycle` has completed this, work flow control is event-driven. The other plugins, and indeed the `IManagerLifeCycle` plugin, are called in response to third parties binding services, worker checks, etc. These plugins are most likely to call other plugins to help in their work.

3.4 Key plugin principles

Several ideas to help us get a feel for plugins.

3.4.1 The plugin manager and lookup

A plugin manager allows plugins to be added (i.e. plugged in) using its `add()` method. Those plugins which have been added can then use the manager to look up other plugins and talk to them.

When a plugin is added it may be given a string name. It is also given an initialisation parameter and told whether to initialise immediately or to wait until needed.

A lookup is achieved by matching a plugin name or its type (i.e. class type or interface). The plugin manager ensures each matching plugin is initialised successfully before it can be returned from a `lookup()` method call. So if it isn't already initialised it happens then, and if the initialisation fails then that plugin is removed from the plugin manager and not returned.

3.4.2 Adding by class and by instance

We can add a plugin to a plugin manager either by specific instance or by class.

Adding by class means we just say what kind of plugin we're adding. If it's never needed it may never get instantiated. This is the truly modular approach because the plugin cannot be pre-configured. If it needs to wire itself up to other plugins when it starts then it almost certainly has to go via the plugin manager. It also requires one less line of code.

Adding by instance means we construct a new specific instance of the plugin and add that. This is perhaps more intuitive but may lead to bad habits.

3.4.3 Initialisation

Before a plugin can be used the following happens, courtesy of the plugin manager. But each step only happens once per plugin.

1. If it was added by class then a new instance is constructed using its default constructor.
2. The plugin is initialised via its `init()` method, which returns a success flag.

If either step fails then the plugin manager removes the plugin and it is not used again.

It is worth stressing that initialisation happens no more than once per plugin. This is regardless of how many interfaces and jobs it has.

3.4.4 Eager initialisation

Usually a plugin manager will initialise a plugin only when it has to—i.e. when it matches a plugin lookup search and has to return an array of results.

But we can force a plugin to be initialised when it is added to a plugin manager. This is called *eager initialisation*, because it probably doesn't need to be used yet.

If eager initialisation fails the `add()` method, which invoked the initialisation, throws an exception. But if initialisation is not eager—i.e. it occurs from a lookup—then failure will be handled quietly and the `lookup()` method will just not return that failed plugin (although it may return others).

3.4.5 Shutdown

The plugin manager will shutdown a plugin before it removes it. The plugin's `shutdown()` method is called only if it has been successfully initialised.

3.4.6 Identifying plugins

Each plugin can be given a `String` name, and every plugin is of a certain type (i.e. class). This is how we identify them, but no uniqueness is enforced.

Any object which implements the general `IPlugin` interface is a plugin. However, to be useful it almost certainly also needs to implement an interface for a specific plugin, and this is how its users will recognise it.

For example, the skeleton application's `LifeCyclePlugin` implements not only `IPlugin` but also `IManagerLifeCycle`. Implementing `IPlugin` means it can be plugged in, initialised, and shutdown. But implementing `IManagerLifeCycle` means it will be called on for the specific purpose of reacting to manager life cycle events.

3.4.7 Multiple plugins of the same type

Sometimes several plugins of the same type are expected to co-exist simultaneously, and sometimes not.

For example, there may be several plugins which have resource requirements for worker netlets. If each is added to the plugin manager then they will all be queried so their requirements can be aggregated. Here's how it might work:

1. Plugin manager P is a `WorkerPool` . It organises workers.
2. Plugin A would like workers to have access to port 53. It implements `IWorkerResourceRequester` and gets added to P.
3. Plugin B would like workers to have access to any port over 1024. It also implements `IWorkerResourceRequester` and also gets added to P.
4. Plugin C would like workers to have file space of 2MB or more. It implements `IWorkerResourceRequester` and gets added to P.
5. Some time later the worker pool P wants to start a new worker. It looks up every plugin that implements `IWorkerResourceRequester` and finds A, B and C. It asks every one what its requirements are, then goes to look for a node which has all those resources.

In this example the worker pool operates with several plugins all of which implement the same interface.

3.4.8 Some plugins are expected only once

In contrast to the above, some kinds of plugin are only expected to have exactly one instance each.

For example only one plugin is expected to deal with an application cold start. That is, the `Manager` expects exactly one instance of an `IManagerLifeCycle` plugin—no more, and no less. Several such plugins could be added to the plugin manager, but only the first such will be used.

The `IManagerLifeCycle` plugin has an inner class with a handy method to help us here. The inner class `Get` has a static method `one()` which gets the one instance of that plugin. Thus we can call `IManagerLifeCycle.Get.one()` to get the one manager lifecycle plugin. Other plugin types, which also expect just one instance, usually also have a `Get.one()` method.

3.4.9 One plugin can have multiple types

There is no problem for a single plugin to be of several plugin types. This is just a case of a class implementing several interfaces.

For example, look at `SkeletonWorkerManager` (Section 4.6). It implements `IWorkerLifeCycle` which means it responds to workers starting and stopping. But it also implements `IWorkerResourceRequester`, so it can also be queried for resource requirements of workers which are about to be started.

3.4.10 Nested plugins

A plugin manger can have plugins added to it, and these plugins may themselves be plugin managers.

The `ServiceManager` class is an example of this (Figure 2.2). A `ServiceManager` plugin B can be added to a plugin manager A. Then we can add individual service provider plugins C1, C2, etc, to the `ServiceManager` B. When the `ServiceManager` is asked for a particular service it will scan its service providers C1, C2, etc.

3.5 Using plugins

Using a plugin manager we can add, remove and lookup plugins.

3.5.1 The IPluginManager interface

This interface is implemented by anything which wants to be a plugin manager. It is implemented by `PluginManager` as well as `ServiceManager` and `WorkerPool`, which also accept plugins of their own.

```
public interface IPluginManager
{
    /**
     * Add plugin by class. An instance will be contructed only if needed on lookup
     * or eager-initialised.
     * @param name  Name of plugin. If this is null the plugin
     *              is considered to have no name.
     * @param type  Type of plugin. Must implement IPlugin.
     * @param argument_data  Argument data to pass to the init method of the IPlugin.
     * @param eager_initialise If true, initialise plugin on add
     * @throws BeatrixPluginException  If the <code>type</code> does not
     *             implement IPlugin, or initialisation fails in some way.
     */
    public void add(String name, Class type, Object argument_data, boolean eager_initialise)
        throws BeatrixPluginException;

    /**
     * Shortcut for <code>add(null, type, null, eager_initialise)</code>.
     */
    public void add(Class type, boolean eager_initialise)
        throws BeatrixPluginException;
```

```
    /**
     * Add specific instance of a plugin.
     * @param name  Name of plugin. If this is null the plugin
     *              is considered to have no name.
     * @param type  Type of plugin. Must implement IPlugin.
     * @param argument_data  Argument data to pass to the init method of the IPlugin.
     * @param eager_initialise If true, initialise plugin on add.
     * @throws BeatrixPluginException  If initialisation fails in some way.
     */
    public void add(String name, IPlugin plugin, Object argument_data, boolean eager_initialise)
        throws BeatrixPluginException;
    /**
     * Shortcut for <code>add(null, plugin, null, eager_initialise)</code>.
     */
    public void add(IPlugin plugin, boolean eager_initialise)
        throws BeatrixPluginException;
    /**
     * Remove all matching plugins. Each such plugin will have
     * its shutdown method called if it was previously
     * initialised successfully.
     * @param name   Required name for matching plugins. If null
     *               then plugin names are not tested.
     * @param type   Required type for matching plugins. The plugin must
     *               implement or extend this type; it must be it. If null
     *               then plugin types are not tested.
     */
    public void remove(String name, Class type);

    /**
     * Remove all plugins of a particular type.
     * Shortcut for <code>remove(null, type)</code>.
     */
    public void remove(Class type);

    /**
     * Find all matching plugins. Those plugins returned will
     * have been initialised.
     * @param name   Required name for matching plugins. If null
     *               then plugin names are not tested for.
     * @param type   Interface the matching plugins have to implement.
     *               If null then plugin types are not tested for.
     * @return All plugins found with this criteria. Will be a
     *         zero-length array if no matches.
     */
    public Object[] lookup(String name, Class type);

    /**
     * Find all plugins of the specified type.
     * Shortcut for <code>lookup(null, type)</code>.
     */
    public Object[] lookup(Class type);

    /**
     * Find all plugins with the specified name.
     * Shortcut for <code>lookup(name, null)</code>.
     */
    public Object[] lookup(String name);

    /** Shutdown and remove all plugins. Only those plugins
     * which have previously been successfully initialised
     * have their shutdown method called. */
    public void shutdown();
}
```

3.5.2 Adding plugins

Alternative ways of adding plugins:

```
try
{
```

```
    // There are two ways to add by class

    plugin_manager.add("Plugin 1", SomePlugin.class, an_object, true);
    plugin_manager.add(ASecondPlugin.class, false);

    // There are two ways to add by instance

    plugin_manager.add("Plugin 3", specific_plugin, another_object, false);
    plugin_manager.add(fourth_plugin, true);
}
catch (BeatrixPluginException e)
{
    // Deal with a failed add
}
```

The first two plugins are added by class. The first one is given a specific name (*Plugin 1*), an arbitrary object which will be given to its `init()` method, and it is true that it will be initialised right away (eager initialisation).

The second plugin is added more simply. It will have no name, no initialisation parameter (actually, the argument will be `null`), and the `false` indicates it will only be initialised on a lookup. If it never matches a lookup on the plugin manager then it will never be initialised.

The third and fourth plugins are added by instance. *Plugin 3* has that name and is again given an arbitrary object for its `init()` method. It will not be initialised until needed, however.

The fourth plugin has no name and no init argument (actually, a `null` init argument), but it will be initialised now.

Adding plugins can lead to a `BeatrixPluginException`, either if initialisation fails or if we add a class which is not an `IPlugin`.

3.5.3 The order of plugins

Plugins will always be called in the order in which they were added to the plugin manager, because that is the order in which the `lookup()` method will return them.

For example, suppose several plugins register an interest to choose resources. Then whenever resource needs are gathered, those plugins which were added to the plugin manager first will get first pick. Or suppose a `Manager` needs to reference "the" manager life cycle plugin. It will call the first registered manager life cycle plugin, regardless of how many plugins claim to be one.

Very strictly speaking such first-come-first-served selection is at the discretion of any implementation, but we can assume it always to be the case. It would be very unusual for it to be otherwise, and the implementation should certainly document such an oddity.

3.5.4 Using looked up plugins

Here is how we add nested plugins. This is adapted from the `LifeCyclePlugin` class of our skeleton application (Section 4.5):

```
try
{
    String WORKER_POOL = "worker_pool_1";

    pluginManager().add(ServiceManager.class, false);
    pluginManager().add(WORKER_POOL, WorkerPool.class, null, false);

    // Service manager and worker pool need to be initialised before we
    // can use them

    IServiceManager sm
        = (IServiceManager)pluginManager().lookup(IServiceManager.class)[0];
    sm.add("admin", new AdminServiceProvider(), null, false);

    IWorkerPool wp
        = (IWorkerPool)pluginManager().lookup(WORKER_POOL, IWorkerPool.class)[0];
    wp.setRedundancyMin(2);
    wp.setRedundancyMax(4);
    wp.setWorkerType(WORKER_POOL);

    wp.add(InternalFacetCache.class, false);
    wp.add(SkeletonWorkerManager.class, false);
}
catch(BeatrixPluginException e)
{
    // Deal with add failure
}
```

This piece of code is in a subclass of `AbstractPlugin` whose `pluginManager()` method just gets the plugin manager. We see that we add a `ServiceManager` plugin and a `WorkerPool` plugin. Neither is initialised, and their `init()` methods don't need a parameter object.

Then we want to add a plugin to the service manager. The service manager is a plugin which acts as a collection point for all the services provided by this netlet. It is also a plugin manager itself. We want to add a plugin called `AdminServiceProvider` by which our application owner can control the application. So we (a) lookup the service manager plugin. Only this way will we get an initialised instance of it. Then we (b) add the desired service provider plugin. We add it under a specific name, *admin*. See Section 5.8 for more on how services are implemented.

The `WorkerPool` plugin is also a plugin manager. By adding plugins to it we define a specific kind of worker netlet—in this case the *worker_pool_1* netlet. Again we (a) look it up and (b) add plugins to it. But in between we set various criteria for that worker pool: that there should be between 2 and 4 workers at any one time and that it is called *worker_pool_1*. See Section 5.6 for more on managing workers.

Here is an example of using the entire array returned by lookup, rather than just the first element. It is adapted from code inside the `Manager` class and shows what happens when it gets a worker-started event. It looks for all those plugins which implement the `IWorkerLifeCycle` interface, because that indicates an interest in workers' life and death, and calls their `workerStarted()` methods in turn:

```
// Variable worker is the netlet which has just started

Object[] p = _plugin_manager.lookup(IWorkerLifeCycle.class);

for(int i=0; i<p.length; i++)
{
    ((IWorkerLifeCycle)p[i]).workerStarted(worker);
}
```

3.5.5 Removing plugins

This is just a case of identifying what needs to be removed. Internally, a plugin will be shutdown before it is removed:

```
// Remove all plugins with this name and which implement this interface
pluginManager().remove("worker_pool_1", IWorkerPool.class);

// Remove all plugins which implement this interface
pluginManager().remove(IWorkerPool.class);

// Remove all plugins with this name
pluginManager().remove("worker_pool_1", null);
```

As above, we assume the method `pluginManager()` returns the manager from which we want to remove the plugins.

3.6 Writing a plugin

The code in this section is adapted from the `AdminServiceProvider` plugin of the skeleton application (Section 4.3).

3.6.1 A complete and simple plugin

Here is the complete code to a simple plugin. It will be explained in the sections that follow.

```
package com.mycompany.jtrix.plugins;

import org.jtrix.base.*;

import org.jtrix.facets1.util.properties.Property;
import org.jtrix.project.libjtrix.netlet.FacetCollection;
import org.jtrix.project.beatrix.plugins.*;
import org.jtrix.project.beatrix.plugins.util.ISimpleServiceProvider;

/**
 * A plugin to provide an "admin" service for the application owner.
 * Should be added to an IServiceManager, which is a plugin manager.
 */

public class AdminServicePlugin implements IPlugin, IServiceProvider
{
    private IPluginManager _plugin_manager;

    // Two methods to implement IPlugin

    public boolean init(IPluginManager plugin_manager, Object data)
```

```
    {
        _plugin_manager = plugin_manager;
        return true;
    }

    public void shutdown()
    {}

    // Two methods to implement ISimpleServiceProvider

    public IFacetCollection createFacetCollection(String service_type, Property args)
    {
        return new FacetCollection();
    }

    public boolean isAdminProvider()
    {
        return true;
    }
}
```

3.6.2 Code structure

The class needs to implement the `IPlugin` interface to be a plugin. This is just the `init()` method and the `shutdown()` method. It also implements two methods for the `ISimpleServiceProvider` interface.

By implementing `ISimpleServiceProvider` it can be added to a `ServiceManager` which acts as a collection point for all plugins which provide services and which is itself a plugin manager.

Thus every plugin implements `IPlugin`, to enable it to be added to a plugin manager, and at least one other interface to perform its specific task.

3.6.3 The IPlugin interface

As found in `org.jtrix.project.beatrix.plugins`:

```
public interface IPlugin
{
    /**
     * Initialise the plugin before first needed.
     * Here, "first needed" is when it successfully
     * matches against a lookup on its plugin manager.
     * Should only be called by the plugin manager which will
     * only call it once.
     * If initialisation fails the plugin manager will
     * remove this plugin.
     *
     * @param plugin_manager Plugin manager that holds this plugin.
     * @param data   Arbitrary argument data supplied when it
     *               was originally added to the plugin manager.
     * @return  Success flag. If false then the plugin will be
     *          removed from its plugin manager.
     */
    public boolean init(IPluginManager plugin_manager, Object data);

    /**
     * Shutdown the plugin. Should only be called by the plugin
     * manager, which will only call it if was previously
     * successfully initialised.
     */
```

```
    public void shutdown();
}
```

Recall from Sections 3.4.3–3.4.5 how plugin initialisation and shutdown occur: initialisation will occur some time before first use, and possibly when it is added to its plugin manager; shutdown will only occur after a successful initialisation, just before it is removed from its plugin manager.

Recall also (Section 3.5.2) our various ways of adding plugins.

Thus our `IPlugin.init()` method above may be called immediately on add, depending on whether it was added with the "eager initialisation" flag as true. Certainly it will be called before it is first returned by a lookup on its plugin manager.

When `init()` is called its two parameters are the plugin manager itself and the `Object` (if any) specified when it was added. Our implementation above saves the plugin manager reference (even though we don't use it) as it can be useful if we want to lookup other plugins via that plugin manager.

Note that if the `init()` method returns `false` then it will be removed from its plugin manager; the `shutdown()` method will not be called in this case.

3.6.4 Other plugin interfaces

Any other interfaces a plugin implements are entirely free. Several are defined by Beatrix. A plugin may implement any number of these or others.

If we implement a pre-defined plugin interface (e.g. `ISimpleServiceProvider`) then as long we add it in the right place (e.g. an `ISimpleServiceProvider` expects to be added to the `ServiceManager` implementation) then it will be called as necessary. If we implement our own plugin interface then we just need to ensure our own code calls it as necessary. See Chapter 5 on key plugin interfaces.

Note however that these interfaces should not extend `IPlugin`. The `IPlugin` interface is reserved for the implementation and is a way for the plugin manager to initialise it and shut it down.

3.6.5 AbstractPlugin

`AbstractPlugin` can be found in `org.jtrix.project.beatrix.plugins.util`.

The `AbstractPlugin` class is a simple implementation of `IPlugin`, and the code above could have extended this class and hence have its `init()` and `shutdown()` methods ready-made.

`AbstractPlugin` also supplies some other methods, notably `pluginManager()` which gives direct reference to its plugin manager. Note, however, that `pluginManager()` only gives the right return value after the `AbstractPlugin.init()` method has been called. So if we want to override the `AbstractPlugin.init()` method we usually begin like this:

```
public boolean init(IPluginManager plugin_manager, Object data)
{
    if (!super.init(plugin_manager, data))
    {
        return false;
    }

    // Now we can call pluginManager() throughout this class.
    // Rest of our init code goes here...
}
```

Chapter 4

A simple Beatrix application

Here's how we use Beatrix to write and use a simple distributed application which will serve a hello world message to a console. The principles of plugins have been covered in Chapter 3 but see Chapter 5 for the deeper details of all this code.

The application will:

- Provide an administation service through the managers, allowing the application owner to control it. This service will be called *admin*.
- Provide a consumer service through workers, allowing them to serve the hello world message to any console that connects to them. This service will be called *skeleton*.
- Maintain redundancy by keeping 2–4 managers alive at all times.
- Maintain redundancy by keeping 2–4 workers alive at all times.
- Demand network and disk resources for workers (though it doesn't use either).
- Have managers adding plugins to new workers remotely.

Additionally Beatrix does the following for us:

- Maintains redundancy by ensuring each worker runs on a different node.
- Maintains redundancy by ensuring each manager runs on a different node.
- Allows unused resources to be released or re-used as necessary.

- Allows launch and management through a command line console.

We need to implement five plugins and one interface—see Figure 2.2 for a visual overview. The interface:

- `ISkeletonFacet`. An interface that contains the `getMessage()` method, providing the hello world message.

The plugins:

- `AdminServiceProvider`. Provides the owner's administration service. We will call the service *admin*.
- `SkeletonServiceProvider`. Provides the basic consumer service to deliver the hello world message. We call this service *skeleton*.
- `LifeCyclePlugin`. Handles the managers' life cycles: cold starts, warm starts, joining and leadership changes. Program flow starts here.
- `SkeletonWorkerManager`. A plugin to the managers which sets up the workers. Sets their resource requirements and remotely adds the plugins they need.
- `SkeletonWorker`. The main worker plugin. It binds the resources allocated to it but doesn't use them.

An additional class, `SkeletonFacet`, is included in the *SkeletonWorker.java* file. It implements the `ISkeletonFacet`.

4.1 How to read this code

The skeleton code that follows is not meant to be understood first time; there are many concepts we have not explained. However, do read it to get a feel for what happens in an application.

The plugins are presented easiest-first, but flow control begins in the `LifeCyclePlugin` (Section 4.5). Exactly which method is called depends on the state of the application: is this a cold start or a warm start?

The rest of the document explains all the Beatrix concepts, always referring back to this skeleton application, and hence building up the full picture from scratch. At the end you will not only be able to return to the skeleton code and understand it, but also know enough concepts, classes and interfaces to write a good-sized Beatrix application of your own.

4.2 ISkeletonFacet

The facet we need is simply a message provider:

```
package org.jtrix.project.skeleton2.facets;

import org.jtrix.base.*;

/**
 * Simple user interface facet.  Applications implement this to
 * provide a very simple user interface for their applciations.
 *
 * @version     $Revision: 1.3 $
 * @author      ulf@jtrix.org
 * @author      nadia@jtrix.org
 */

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

4.3 AdminServiceProvider

This class is a plugin which provides the admin service for the application's owner. Its superclass, `AbstractPlugin`, provides the basic methods to make this a plugin. The methods here provide its specific functionality. This will be plugged into a `ServiceManager` which provides default admin service tools; this classes does not provide anything additional, but does confirm that those tools should be provided.

```
package org.jtrix.project.skeleton2.plugins;

import org.jtrix.base.*;
import org.jtrix.facets1.util.properties.*;
import org.jtrix.project.libjtrix.netlet.FacetCollection;
import org.jtrix.project.beatrix.plugins.*;
import org.jtrix.project.beatrix.plugins.util.*;

/**
 * Skeleton's admin service, which provides just the standard Beatrix
 * admin facets.
 *
 * @author  ulf@jtrix.org
 * @author  nadia@jtrix.org
 * @version $Revision: 1.13 $
 */
public class AdminServiceProvider
    extends AbstractPlugin implements ISimpleServiceProvider, IPlugin
{
    /**
     * Create a facet collection for this service label, called once
     * for each service bind request.
     * @param session  Service session from client
     * @param service_type  Service name requested by client
     * @param args  Property argument pulled from client's warrant
     */
    public IFacetCollection createFacetCollection(String service_type, Property args)
    {
        return new FacetCollection();
    }

    /** @return True if this service provider is an admin service.
     */
```

```
    public boolean isAdminProvider()
    {
        return true;
    }
}
```

We don't need to do much—we only want some standard admin facets. Since we have flagged it as an admin service we will get admin facets added to the (empty) facet collection we return.

See Section 5.8 for more on services.

4.4 SkeletonServiceProvider

The consumer service plugin is the inverse of the `AdminServiceProvider`—our basic facet collection is non-empty (it contains an implementation of `ISkeletonFacet`, plus the same thing wrapped as a console facet) and it is not an admin service. See Section 5.8 for more on services and Section 6.4 for more on implementing console interfaces. See Section 7.5.2 for some details on the `Debug` class.

```
package org.jtrix.project.skeleton2.plugins;

import org.jtrix.base.*;
import org.jtrix.facets1.util.properties.*;
import org.jtrix.facets1.service.common.*;
import org.jtrix.project.libjtrix.netlet.*;
import org.jtrix.project.libjtrix.debug.*;
import org.jtrix.project.beatrix.common.*;
import org.jtrix.project.beatrix.plugins.*;
import org.jtrix.project.beatrix.plugins.util.*;
import org.jtrix.project.skeleton2.facets.ISkeletonFacet;

/**
 * Provides a service to be plugged into the SkeletonWorker.
 *
 * @author  ulf@jtrix.org
 * @author  nadia@jtrix.org
 * @version $Revision: 1.14 $
 */
public class SkeletonServiceProvider
    extends AbstractPlugin implements ISimpleServiceProvider, IPlugin
{
    /**
     * Create a facet collection for this service label, called once
     * for each service bind request.
     * @param session  Service session from client
     * @param service_type  Service name requested by client
     * @param args  Property arguments from client's warrant
     */
    public IFacetCollection createFacetCollection(String service_type,Property args)
    {


        FacetCollection fc = new FacetCollection();
        fc.addFacet( ISkeletonFacet.class.getName(), new SkeletonFacet() );

        try
        {
            IConsoleFacet my_console = new ConsoleFacetFactory(ISkeletonFacet.class)
                .createConsoleFacet(new SkeletonFacet());
            fc.addFacet( IConsoleFacet.class.getName(), new JointConsoleFacet(
                my_console, peerSupport().getStandardConsole()) );
        }
```

```
        catch(ConsoleFacetFactory.InvalidFacetException e)
        {
            // Our skeleton facet is not suitable to be made into a console
            Debug.exc(this,e,"ISkeletonFacet can't be made into a console");
        }
        catch(ConsoleFacetFactory.InvalidImplementationException e)
        {
            // If SkeletonFacet is not an implementation of ISkeletonFacet
            Debug.exc(this,e,"SkeletonFacet doesn't implement ISkeletonFacet");
        }

        return fc;
    }

    /**
     * @return true if this Service Provider is an admin service.
     */
    public boolean isAdminProvider()
    {
        return false;
    }

    /** An implementation of ISkeletonFacet, which is also an
     * ICommunicationServer and hence can respond to connection loss.
     */
    static class SkeletonFacet implements ICommunicationServer, ISkeletonFacet
    {

        public void connectionClosed()
        {
            // Respond to connection loss
        }

        public String getMessage()
        {
            return "Hello, skeleton plugin world";
        }
    }
}
```

4.5 LifeCyclePlugin

This is where we handle manager (and hence application) life cycle changes. Flow control begins here, with the `coldStart()` method; when this returns initialisation is essentially complete, and all other action is event-driven.

This is an essential plugin for any application as it helps it start. In this example the bulk of the work is the manager adding plugins to itself. See Section 5.2 for more on application start-up, Section 5.6 on worker pools and Section 5.7 on resources.

```
package org.jtrix.project.skeleton2.plugins;

import org.jtrix.base.*;
import org.jtrix.facets1.util.properties.*;
import org.jtrix.project.libjtrix.debug.*;
import org.jtrix.project.beatrix.common.*;
import org.jtrix.project.beatrix.plugins.*;
import org.jtrix.project.beatrix.plugins.hooks.*;
import org.jtrix.project.beatrix.plugins.util.*;

import java.io.Serializable;

/**
```

```
 * @author  ulf@jtrix.org
 * @author  nadia@jtrix.org
 * @version $Revision: 1.28 $
 */
public class LifeCyclePlugin
    extends AbstractPlugin implements IManagerLifeCycle, IPlugin
{
    static final String WORKER_POOL = "worker_pool_1";

    IWorkerPool workerPool()
    {
        return (IWorkerPool)pluginManager().lookup(WORKER_POOL, IWorkerPool.class)[0];
    }

    public boolean init(IPluginManager plugin_manager, Object data)
    {
        Debug.set("skeletonmanager",-1,0);

        if (!super.init(plugin_manager, data))
        {
            return false;
        }

        try
        {
            Debug.msg(Debug.INFO, this, "init(): plugin_manager: ", pluginManager().toString());
            Debug.msg(Debug.INFO, this, "Adding ServiceManager.class");

            pluginManager().add(ServiceManager.class, false);
            pluginManager().add(WORKER_POOL, WorkerPool.class, null, false);

            // service manager needs to be initialised before add can be called
            IServiceManager sm
                = (IServiceManager)pluginManager().lookup(IServiceManager.class)[0];
            sm.add( "admin", new AdminServiceProvider(), null, false);

            IWorkerPool wp = workerPool();
            wp.setRedundancyMin(2);
            wp.setRedundancyMax(4);
            wp.setWorkerType(WORKER_POOL);

            wp.add(InternalFacetCache.class, false);
            wp.add(SkeletonWorkerManager.class, false);

            // Add a FileSystemLocator plugin to the worker pool so that it
            // always demands file system resources for its workers. We give
            // it a parameter, a LocatorInfo, which describes the file system
            // requirements: 1MB of non-redundant disk space.

            FileSystemLocator.LocatorInfo info
                = new FileSystemLocator.LocatorInfo(1024, 1024, true, 1, "disk", 5);
            wp.add("disk", FileSystemLocator.class, info, false);

            wp.add(SocketFactoryLocator.class, false);
        }
        catch(Exception e)
        {
            Debug.exc(this, e, "Initialise failed");
            return false;
        }

        return true;
    }

    public Warrant coldStart(IPropertyCollection args) throws BeatrixException
    {
        Debug.msg(Debug.INFO,this,"coldStart()");

        changeLeaderMode(true);

        return managerSupport().getWarrant("admin",null);
    }

    public void warmStart(Serializable arg)
```

```
    {
        Debug.msg(Debug.INFO,this,"warmStart()");

        changeLeaderMode(true);
    }

    public void join(Serializable arg)
    {}

    public void changeLeaderMode(boolean you_are_leader)
    {
        IWorkerPool wp = workerPool();

        if (you_are_leader)
        {
            ILeaderSupport ls = (ILeaderSupport)pluginManager()
                .lookup(ILeaderSupport.class)[0];
            ls.setRedundancy(2,4);

            wp.start(false);
        }
        else
        {
            wp.stop(false);
        }
    }
}
```

4.6 SkeletonWorkerManager

A plugin for the manager which handles both worker resource requirements and their life cycle events.

```
package org.jtrix.project.skeleton2.plugins;

import org.jtrix.project.libjtrix.debug.*;
import org.jtrix.project.beatrix.handle.*;
import org.jtrix.project.beatrix.plugins.*;
import org.jtrix.project.beatrix.plugins.hooks.*;
import org.jtrix.project.beatrix.plugins.util.*;

/**
 * Plugin that manages start up and shutdown of workers.
 *
 * @author  ulf@jtrix.org
 * @author  nadia@jtrix.org
 * @version $Revision: 1.17 $
 */

public class SkeletonWorkerManager extends AbstractPlugin
    implements IWorkerLifeCycle, IWorkerResourceRequester
{
    /** Respond to a new worker starting by inserting plugins remotely.
     */
    public void workerStarted(IWorkerHandle worker)
    {
        try
        {
            Debug.msg(Debug.INFO, this, "workerStarted()");
            IInternalFacetCache ifc = (IInternalFacetCache) pluginManager()
                .lookup(IInternalFacetCache.class)[0];

            // Add a service manager plugin to the worker.
            // Add a plugin for the worker's basic functionality.
            // Add a plugin to the service manager which provides the skeleton gservice.

            IRemotePluginManager pm = (IRemotePluginManager)ifc
                .getInternalFacet(worker, IRemotePluginManager.class.getName(), null, true);
```

```
            pm.remoteAdd(RemoteServiceManager.class, false);
            pm.remoteAdd(SkeletonWorker.class, true);

            Debug.msg(Debug.INFO, this, "workerStarted, looking up service manager");
            IRemoteServiceManager sm = (IRemoteServiceManager)pm
                .remoteLookup(IRemoteServiceManager.class)[0];
            Debug.msg(Debug.INFO, this, "Got service manager");
            sm.remoteAdd("skeleton",SkeletonServiceProvider.class, null, false);
        }
        catch(Exception e)
        {
            Debug.exc(this, e, "workerStarted() failed.");
        }
    }

    /** Respond to a worker dying.
     */
    public void workerDied(IWorkerHandle worker)
    {}

    /** Demand particular resources for a worker. We only do this as a demonstration
     * of how to demand resources. We demand filesystem resources easily
     * with FileSystemLocator.LocatorInfo in the LifeCyclePlugin. Here we demand
     * a socket factory and set our requirements by hand.
     */
    public void configureResourceHandle(IResourceHandle resource)
            throws IWorkerResourceRequester.ResourceUnavailableException
    {
        ISocketFactoryLocator sfl
            = (ISocketFactoryLocator)pluginManager().lookup(ISocketFactoryLocator.class)[0];
        IResourceClassHandle[] net_classes = sfl.locateByCount(1, 1);

        if(net_classes.length == 0)
        {
            throw new IWorkerResourceRequester.ResourceUnavailableException(
                "No resource classes");
        }

        try
        {
            resource.addBinding("net", 10, net_classes, null);
        }
        catch (Exception e)
        {
            Debug.exc(this,e,"Resource creation failed");
            throw new IWorkerResourceRequester.ResourceUnavailableException(
                "Resource creation failed");
        }
    }
}
```

A worker starting is a prompt to add plugins remotely; a worker dying requires no action. Meanwhile, the plugin also sets worker resource requirements for networking; worker disk requirements are handled entirely by the `FileSystemLocator`.

See Section 5.4 on internal facets, Section 5.9 on remote plugin management and Section 5.7 on resources.

4.7 SkeletonWorker

The main plugin for a worker, which binds its allocated resources, even though it doesn't use them. See Section 5.7.2 for more on using resources.

```
package org.jtrix.project.skeleton2.plugins;

import org.jtrix.base.*;
import org.jtrix.project.libjtrix.debug.*;
import org.jtrix.project.beatrix.plugins.*;
import org.jtrix.project.beatrix.plugins.hooks.*;
import org.jtrix.project.beatrix.plugins.util.*;

/**
 * Basic skeleton worker functionality. It binds its resources, just as a demo,
 * although it doesn't do anything with them.
 *
 * @author  ulf@jtrix.org
 * @author  nadia@jtrix.org
 * @version $Revision: 1.10 $
 */

public class SkeletonWorker extends AbstractPlugin implements IPlugin
{
    private IService _net_service;
    private IService _disk_service;

    public boolean init(IPluginManager plugin_manager, Object data)
    {
        try
        {
            Debug.set("skeletonworker",-1,0);

            super.init(plugin_manager, data);
            IWorkerResourceProvider wrp = (IWorkerResourceProvider)pluginManager()
                .lookup(IWorkerResourceProvider.class)[0];
            IPeerSupport ps = (IPeerSupport)pluginManager().lookup(IPeerSupport.class)[0];
            _net_service = wrp.bindResource("net", ps.createClientSession(false));
            _disk_service = wrp.bindResource("disk", ps.createClientSession(false));

            Debug.msg(Debug.INFO, this, "Skeleton worker started");
        }
        catch (Exception e)
        {
            Debug.exc(this, e, "Init failed");
            return false;
        }

        return true;
    }
}
```

4.8 Compiling the application

To compile our application we need to include the following JARs on our classpath:

- *jtrix.jar* because we use basic Jtrix constructs such as `IService` and `IRemote`.
- *facets1.jar* because we use "standard" facets and related classes, including `IConsoleFacet` and `Property`.
- *libjtrix.jar* because we use `Debug`, `ConsoleFacetFactory` and others.
- *beatrix.jar* for Beatrix itself.

Were we to write a more complex application we might of course need more JARs.

We compile the whole into a single JAR file which we name *skeleton2.jar*. We can optionally include a manifest with a line like this:

```
jar-label: skeleton2
```

This labels the JAR for SAS, giving it a handy way of referencing it in future. If we omit this then it will take the JAR's filename as a label instead. However, giving a JAR label explictly is a good way of ensuring it gets a helpful name we know about. JARs with the same label are considered to be the same JAR, although a JAR hash code is used to double-check, and helps prevent Jtrix fetching JARs it already has. See also Section 6.3.1.

4.9 Creating the launch descriptor

Once our application is compiled we need to create a *launch descriptor*, which is an XML file that allows us to launch the application onto a remote node. We use the *jtrixmaker* tool:

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

The parameters are detailed more in Appendix B, but meanwhile we have this:

–type What type of thing we're making. In this case a launcher descriptor for a Beatrix application.

–outfile What the resulting file is to be called.

–x500dn AnX.500 distinguished name for the application. Any string such as *name1=value,name2=value2,...* is suitable. Here, O is Organisation, CN is canonical name. See Section 7.1 on X.500 names.

–jardirs What directories *jtrixmaker* should look in to find the required JARs. Our main Jtrix JARs and *skeleton2.jar* are all in */usr/lib/jtrix* (this might be different on your system) so we name that. We can specify several JARs here if needed.

–jars A series of whichever JAR files the application will need in order to run. In this case they are: *skeleton2.jar*, *beatrix.jar* and *facets1.jar*, just as we needed for compilation, and Beatrix additionally needs the XML JARs *jaxp.jar* and *parser.jar*. However, note that we don't need *jtrix.jar* in the launch descriptor. This is always the case, because we can always count on it being there on whichever node we happen to be running on.

–access Some arbitrary label for our access points. This simply tells *jtrixmaker* that all access points are one particular kind of netlet, with this arbitrary label. It ties closely with. . .

–codebase Says what JARs are needed for this particular kind of netlet. The first argument is the access point label we just gave. The rest of the arguments say what JARs it needs (all the JARs we gave before except the XML ones). The upshot is that when we connect to our application we get an access point netlet which needs fewer JARs. Thus is loads quicker. See Section 7.4.

–plugins A series of one or more plugins, which Beatrix will try to add as classes, but not initialise. At least one of these should be the `IManagerLifeCycle` plugin. On launch the `Manager` will do a lookup for an `IManagerLifeCycle` plugin and will call the first one it finds for a cold start. In short, if we only list an `IManagerLifeCycle` plugin, and that adds all the manager's other plugins, then that will be fine.

The result is something like this:

```
<!DOCTYPE application PUBLIC "-//jtrix.org//TEXT application-0.1//EN"
    "http://www.jtrix.org/dtd/application-0.1.dtd">
<application>
    <jar jar_id='beatrix'>
        <url>file:/usr/lib/jtrix/beatrix.jar</url>
    </jar>
    <jar jar_id='skeleton2'>
        <url>file:/usr/lib/jtrix/skeleton2.jar</url>
    </jar>
    <jar jar_id='parser.jar'>
        <url>file:/usr/lib/jtrix/parser.jar</url>
    </jar>
    <jar jar_id='libjtrix'>
        <url>file:/usr/lib/jtrix/libjtrix.jar</url>
    </jar>
    <jar jar_id='facets1.jar'>
        <url>file:/usr/lib/jtrix/facets1.jar</url>
    </jar>
    <jar jar_id='jaxp.jar'>
        <url>file:/usr/lib/jtrix/jaxp.jar</url>
    </jar>
    <codebase codebase_id='ap'>
        <element jar_id='beatrix'/>
        <element jar_id='skeleton2'/>
        <element jar_id='libjtrix'/>
        <element jar_id='facets1.jar'/>
    </codebase>
    <codebase codebase_id='DEFAULT_FOR_ALL'>
        <element jar_id='skeleton2'/>
        <element jar_id='parser.jar'/>
        <element jar_id='jaxp.jar'/>
        <element jar_id='beatrix'/>
```

```
        <element jar_id='libjtrix'/>
        <element jar_id='facets1.jar'/>
    </codebase>
    <boot class='org.jtrix.project.beatrix.plugins.netlets.Manager'
          codebase_id='DEFAULT_FOR_ALL'/>
    <cmdline>
        <argument type="string" name="plugins" required="yes">
            org.jtrix.project.skeleton2.plugins.LifeCyclePlugin </argument>
        <argument type="x500dn" name="x500" required="yes">
            O=jtrix CN=skeleton2</argument>
        <argument type="string" name="name" required="yes">
            Initial Manager</argument>
    </cmdline>
    <config>
        <![CDATA[
            ....
        ]]>
    </config>
</application>
```

This launcher descriptor allows us to launch our application and all its JARs (referenced in the XML) onto a remote node. See Section 6.3.1 for more details on this.

See Section 4.10 for how we use the console to run our application.

4.10 Launching and running the application

To launch and run an application we need access to a hosting service and SAS. If you don't have these already there are two options:

1. Run your own node with these services, as explained in "Start running Jtrix" at `http://www.jtrix.org`.
2. Get warrants to use Jtrix.org's own hosting service and SAS. Get them at `http://www.jtrix.org/warrant/hosting-warrant.xml` and `http://www.jtrix.org/warrant/sas-warrant.xml` respectively.

4.10.1 Launching the application

First, we assume the file *hosting-warrant.xml* contains a warrant to a hosting service, and *sas-warrant.xml* contains a warrant to a SAS. Here's how we run the launcher and announce that these are the services we're going to use. If you aren't familiar with the launcher it's explained in "Start running Jtrix" at `http://www.jtrix.org`.

```
% launcher
Initialising Nodality...
launcher> host {warrant:hosting-warrant.xml}
Downloading access point                    [5....4....3....2....1....0]
Downloading access point jars               [5....4....3....2....1....0]
Waiting for netlet to service bind... done
launcher> sas {warrant:sas-warrant.xml}
Downloading access point                    [5....4....3....2....1....0]
Downloading access point jars               [5....4....3....2....1....0]
Waiting for netlet to service bind... done
SAS warrant recognised: will upload jars
launcher>
```

And now here's how we launch the application:

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

4.10.2 Running the application

Once the application's launched we can play with it:

```
launcher> list
sk
launcher> list sk
  Facet: org.jtrix.facets1.service.common.IConsoleFacet
sk.quit - Shut down the application
sk.list-managers - Show all managers and their nodes
sk.list-workers - Show all workers and their nodes
sk.get-warrant - <type> [<service-data>] Get refreshed warrant
sk.set-manager-redundancy - <min> <max> Set the manager redundancy
sk.list-management-state - Display the management state
launcher> dump $sk skadmin.xml
launcher> bw=‘sk.get-warrant skeleton‘
launcher> connect skbasic $bw
Downloading access point                     [5....4....3....2....1....0]
Downloading access point jars                [5....4....3....2....1....0]
Waiting for netlet to service bind... done
launcher> list
skbasic
sk
launcher> list skbasic
  Facet: org.jtrix.project.skeleton2.facets.ISkeletonFacet
  Facet: org.jtrix.facets1.service.common.IConsoleFacet
skbasic.getMessage - ( Return: java.lang.String )
launcher> skbasic.getMessage
Hello, skeleton plugin world
launcher> echo $skbasic
<<warrant>>
launcher> dump $bw skbasic.xml
launcher> quit

%
```

After launching we are connected automatically to the *admin* service, which we set in our manager netlet code. From here we generate a warrant for the consumer-level *skeleton* service (which we also defined in our code, and plugged in remotely). Then we connect to that service. At this point we are connected to both services, and we get the message from the *skeleton* service. Finally we dump our warrant into a file so we can use it again, or give it to someone else.

All this is just an example. The rest of this document explains the principles behind this application and therefore how to write our own larger ones.

Chapter 5

Key plugins and interfaces

Writing a Beatrix application is a combination of implementing given plugin interfaces and implementing our own plugins from scratch. In this section we present some of the built-in plugin interfaces and plugins.

5.1 Overview

Of all the plugins and plugin interfaces defined in Beatrix, we have four packages:

org.jtrix.project.beatrix.plugins.hooks Plugin interfaces which are used by the `Manager` and `Worker` netlets. Some of these interfaces are also implemented by `Manager` or `Worker`, whereas we are expected to implement others ourselves.

org.jtrix.project.beatrix.plugins.util Extra plugin interfaces and some useful plugin implementations.

org.jtrix.project.beatrix.plugins.netlets Contains the `Manager` and `Worker` netlet implementations.

org.jtrix.project.beatrix.plugins Basic interfaces `IPlugin` (Section 3.6.3) and `IPluginManager` (Section 3.5.1) plus related implementations.

We will look at the `hooks` and `util` packages; the key classes in the other packages have been covered earlier.

5.1.1 The hooks package

Here are some of the built-in plugin interfaces; all are in `org.jtrix.project.beatrix.plugins.hooks` and all are used by the `Manager` and

`Worker` netlets. Those marked (*) are also provided by the `Manager` netlet, while those marked (**) are provided by the `Worker` netlet. So unless otherwise stated, we should not implement our own plugins of these types. Figure 3.1 also shows this.

IBeatrixSASSupport Allows Beatrix to use a service advertisement service (SAS). Only needs to be provided by an implementation of the SAS.

IDeploymentControl (*) Manages the process of deploying the right netlets on nodes with the right resources. Most applications will not access this directly.

IInternalFacetProvider (*)(**) Provides internal facets for intra-application communication—see Section 5.4.3. Although the `Manager` and `Worker` implement these it is likely we will also want to implement our own application-specific `IInternalFacetProvider` plugins.

ILeaderSupport (*) Functionality to help the current leader (Section 5.3).

IManagementState (*) Helps access the management state available to all managers (Section 5.5.1).

IManagerLifeCycle Deals with managers starting (cold starts, warm starts and joins) and leadership changes. Every application needs to implement one of these otherwise the application cannot start. See Section 5.2.1.

IManagerSupport (*) The core work of managers, including the ability to generate warrants for consumers (Section 5.5.2).

IPeerSupport (*)(**) To help managed netlets be part of their application peer group. Find other netlets, announce service eligilibility and set status. See Section 5.4.2.

IServiceManager Collection point for services. Services register with this and can then be retrieved easily as needed—it is a plugin which is also a plugin manager. Every netlet which wants to provide services needs a service manager and, since this is not necessarily all netlets, Beatrix does not add an `IServiceManager` by default. See Section 5.8.

ITransportProvider Allows us to change Beatrix's internal network communication mechanism.

IWorkerLifeCycle Allows a manager to respond to workers stopping and starting (Section 5.6.4).

IWorkerResourceProvider (**) Allows a worker plugin to access the resources allocated to that worker. See Section 5.7.2.

5.1.2 The util package

We have these plugin interfaces and implementations in `org.jtrix.project.beatrix.plugins.util`:

AbstractPlugin A simple plugin implementation which can be extended to make a useful plugin (Section 3.6.5).

FileSystemLocator Implementation of `IFileSystemLocator` (Section 5.7).

IFileSystemLocator Provides a tool for a manager to find a file system resource for a worker.

IInternalFacetCache An internal facet is a facet provided by one Beatrix netlet to another within the same application. They aid inter-netlet/intra-application communication. This interface defines a tool to cache internal facets. See Section 5.4.4.

IRemoteServiceManager An internal facet which one netlet can provide to another allowing the latter remote access to the former's service manager. See Section 5.9.

IServiceProvider Provides services. Plugs into our `ServiceManager` implementation of `IServiceManager`. See Section 5.8.

ISimpleServiceProvider Just like `IServiceProvider`, but is slightly less complex. See Section 5.8.

ISocketFactoryLocator Defines a tool for a manager to find a socket factory resource for a worker. Once it has a socket factory a worker netlet can make its own network connections. See Section 5.7.

ITimer Timer plugin similar to `java.util.Timer`, except that groups of tasks can be cancelled. See Section 5.10.

IWorkerPool A worker pool is a collection point for plugins which support the deployment and running of workers of a specific type. It is added to a manager netlet. See Section 5.6.

IWorkerResourceRequester Allows a plugin to specify resources required by workers. See Section 5.7.5.

InternalFacetCache An implementation of `IInternalFacetCache` (Section 5.4.4).

PluggablePluginManager A plugin which is also a plugin manager. A `lookup()` method call gets passed to its own plugin manager if there's no match in itself.

RemoteServiceManager An implementation of `IRemoteServiceManager` (Section 5.9).

ServiceManager An implementation of `IServiceManager` in the hooks package (Section 5.8). Each `IServiceProvider` or `ISimpleServiceProvider` plugin added to to a `ServiceManager` provides a service to our application's consumers.

SocketFactoryLocator An implementation of `ISocketFactoryLocator` (Section 5.7).

TimerPlugin An implementation of `ITimer` (Section 5.10).

WorkerPool An implementation of `IWorkerPool`. In this implementation workers from the same pool will always run on different nodes to aid redundancy. See Section 5.6.

5.1.3 Essential plugin summary

Tables 5.1, 5.2 and 5.3 provide a useful quick guide to help you decide what plugins to look in depending what you want to do. The rest of this chapter fills in the details.

5.2 Application start-up

Recall the three modes of manager start-up: cold start, warm start and join (Section 2.1.5) and that when the first manager starts—which will be either a cold start or a warm start—it must start up the other managers and the first workers.

5.2.1 IManagerLifeCycle

To achieve application start-up, every application must implement an `IManagerLifeCycle` plugin. This is a plugin interface that Beatrix expects but cannot implement itself:

```
public interface IManagerLifeCycle
{
    /**
     *  Applications must implement this to define cold start behaviour.
     *  ColdStart is the primary boot process initiated by the
     *  application administrator.
     *
     *  @param args  Application specific properties defined
     *               by the bootstrap process,
     *               including perhaps the console commandline on launch.
     *  @return      A warrant which will be made available to the application
     *               owner. If this application is to be launched via the
     *               launcher then the warrant's service should provide an
     *               IConsoleFacet.
     *  @throws BeatrixException  If the application cannot cold-start.
     */
    public Warrant coldStart(IPropertyCollection args) throws BeatrixException;

    /**
     *  Applications must implement this to define warm start behaviour.
     *  Will be called by the hosting service when it notices the application
```

What do you want to do?	Where to look/how to do it
Application start and stop	
Cold-start the application.	`IManagerLifeCycle`
Warm-start (after failure).	`IManagerLifeCycle`
Act when this netlet is a new manager joining the existing ones.	`IManagerLifeCycle`
React when the leader changes.	`IManagerLifeCycle`
Controlling warm-start and join parameters.	`ILeaderSupport`
Set the number of managers.	`ILeaderSupport`
Close down the application.	`IManagerLifeCycle`
Leadership	
Check if I'm the leader	`ILeaderSupport`
Find the current leader	`IPeerSupport`
Comms between managers	
Use a storage space visible to all managers.	Use the management MIB, with `IManagementState`.
Write to the management MIB.	Use `IManagementState`, but only the leader can do this. For complex changes use the `applyChange()` method.
Read the management MIB.	Use `IManagementState`; any manager can do this.

Table 5.1: Actions associated with managers.

What do you want to do?	Where to look/how to do it
Organise workers	
Create a group of workers to perform a type of service.	Use a worker pool, managed from a manager netlet.
Create a worker pool.	Add a `WorkerPool` plugin to a manager and set the worker pool's parameters.
Set what resources are required for workers.	Add an `IWorkerResourceRequester` plugin to a worker pool.
Respond to workers starting and stopping.	Add an `IWorkerLifeCycle` plugin to a worker pool
Start a worker.	You don't. Workers are started automatically by the worker pool plugin. However, you can increase the number of workers generally by setting the worker redundancy range with `IWorkerPool`.
Stop a worker.	You don't. Workers are stopped automatically by the worker pool plugin. However, you can deccrease the number of workers generally by setting the worker redundancy range with `IWorkerPool`.
Make a worker functional.	Given a basic default `Worker` (1) get its `IRemotePluginManager`, (2) add plugins as needed.
Let a manager find workers, possibly just those who offer a specific service type.	`IManagerSupport`

Table 5.2: Worker-related actions.

What do you want to do?	Where to look/how to do it
Comms within the application	
Get handles to other netlets in the same application.	`IPeerSupport`
Set a status message	`IPeerSupport`
Talk to another netlet in the same application.	Make an interface available as an internal facet. (1) Implement `IInternalFacetProvider`, (2) plug in the implementation, (3) get it remotely using a netlet handle.
Use internal facets more efficiently	`IInternalFacetCache`
Listen for when a connection to a client is cut.	Implement `ICommunicationServer`.
Services	
Make a simple service available.	(1) Plug in a `ServiceManager`, (2) implement an `ISimpleServiceProvider`, (3) plug the service provider into the service manager.
Make a more complex service available.	As above, but implement an `IServiceProvider`.
Create a client service session.	`IPeerSupport`
Create a warrant for anyone: the administrator or a consumer.	`IManagerSupport`
Sign a warrant.	Provide a key pair when you make the application. All warrants will automatically be signed.
Tell workers how to load balance themselves for different services.	`IManagerSupport`
Control a netlet's local environment	
Netlet wants node to terminate it.	`IPeerSupport`
Get netlet's current node.	`IPeerSupport`

Table 5.3: General Beatrix actions.

```
     *  has died.
     *  @param arg   An application specific argument defined using
     *               setWarmStartArgument() from a previous manager.
     */
    public void warmStart(Serializable arg);

    /**
     *  Called when this manager is created by another manager.
     *  @param arg   The argument from the createManager call.
     */
    public void join(Serializable arg);

    /**
     *  Called when the application leadership changes.
     * Not called on start-up, since the leader has not changed.
     */
    public void changeLeaderMode(boolean you_are_leader);

    /** Utility class helping lookup */
    public static class Get
    {
        /** Get the only instance of this plugin in the given manager.
         * Only the first such plugin is returned. If there isn't one then
         * an ArrayIndexOutOfBoundsException is thrown.
         */
        public static IManagerLifeCycle one(IPluginManager pm)
        {
            return ((IManagerLifeCycle)pm.lookup(IManagerLifeCycle.class)[0]);
        }
    }
}
```

A sample implementation is found above in skeleton's `LifeCyclePlugin` (Section 4.5). Notice that this class's `coldStart()` method chooses to call its `changeLeaderMode()` method. This is because, as stated above, Beatrix does not call `changeChangeLeaderMode()` on start-up.

The `coldStart()` method enters with a collection of properties set by the launch descriptor and the administrator on launch. Properties are worth getting to grips with—see Section 7.3 on properties and the next section on the the `IPropertyCollection` argument of `coldStart()`. The `coldStart()` method has to deploy managers and workers. Finally it must also return a warrant to an admin service, so whoever launched the application can control it. That is usually achieved with something like this:

```
IManagerSupport ms = (IManagerSupport)_plugin_manager.lookup(IManagerSupport.class)[0];
return ms.getWarrant("admin", null);
```

which returns a warrant using Beatrix's own `IManagerSupport` plugin, and assumes our admin service is called *admin*. See Section 5.8 for more on services and warrants.

If our `IManagerLifeCycle` plugin extends `AbstractPlugin` then this can be achieved in a single step:

```
return managerSupport().getWarrant("admin", null);
```

Actually, the warrant returned from the `coldStart()` method does not have to be admin warrant; it can be any warrant for a service which

provides an `IConsoleFacet`. This allows the launcher to provide console access to the application. However, since the application administrator is the one who launched the application then we shall in general assume it is an admin warrant.

The `warmStart()` method allows the application to pick up when previously it was running but then died. Its argument is whatever was set via the `ILeaderSupport` plugin (Section 5.3). There is no need to for a warm start to return a warrant, because that was given to the application's administrator when they cold-started it previously. Warm start is just for reviving an application automatically.

The `join()` method is called in a new manager when it joins an already-running application. Its argument is also set via the `ILeaderSupport` plugin (Section 5.3), and again there is no need to return a value.

5.2.2 The coldStart IPropertyCollection parameter

The parameter passed to the `coldStart()` method is just a mapping from `String` names to `Property` objects. These `String/Property` entries are gathered from several sources, including the creation of the launch descriptor XML (Section 4.9):

- The string *plugins* will be mapped to whatever string was specified as the *–plugins* parameter. This entry is listed in the launch descriptor.
- The string *name* will be mapped to a default name for the first manager netlet, and is set by Beatrix. This entry is listed in the launch descriptor.
- Other `String/Property` entries can be specified in the launch descriptor XML. Each entry will have the name, allowed `PropertyType`, whether it is required, and a possible default value. Those required entries without a default value must be specified in the console's *run* command. Those entries with a default value may optionally have that value overridden in the console's *run* command. Any entry not explicitly specified in the launch XML cannot be entered with the *run* command. Jtrixmaker can be used to set these other string/property values.
- The launch descriptor's `config` element is a `BeatrixApplicationDescriptor` object. This also contains `String/Property` pairs which are copied into the `IPropertyCollection` parameter of `coldStart()`. Two are set by default: *debug* and *debug-no* are both long values set to zero. See Section 7.5.2 for more on these.
- If the same string name is set both in the `BeatrixApplicationDescriptor` and the launcher command line, then the command line takes precendence. For example when we use the *run* command we can append *debug={long:–1}* to switch on maximum debug output (Section 7.5.2).

5.3 Being the leader

The `ILeaderSupport` plugin is provided by the `Manager` netlet, and intended to help the current leader.

```
public interface ILeaderSupport
{
    /**
     *  Check if this netlet is the leader.
     * @return   True if this netlet is the leader; false if not.
     */
    public boolean amLeader();

    /**
     *  Set manager redundancy parameters.  If the number falls below
     * <code>min</code>, managers will be created to bring the
     * number of managers up to <code>max</code>. These parameters
     * may be changed mid-application without any problem, though such a
     * change may not have immediate effect. Only has any effect in the
     * current leader.
     */
    public void setRedundancy(int min, int max);

    /**
     *  Set the argument used for application warm starts.  Should allow the
     *  warm start netlet to reconnect or reinitialise the application.
     * @param arg  The argument to pass to the <code>warmStart()</code>
     *     method of the IManagerLifeCycle plugin.
     */
    public void enableWarmStart(Serializable arg) throws BeatrixException;

    /**
     *  Stop the application from being reinitialised.
     */
    public void disableWarmStart();

    /**
     *  Set the argument used for manager joins.
     *  This setting lifes only as long as the current leader.
     *
     * @param arg  serializable argument
     */
    public void setJoinArgument(Serializable arg);

    /** Terminate the application. Only works in the leader */
    public void quitApplication();

    /** Utility class helping lookup */
    public static class Get
    {
        /** Get the only instance of this plugin in the given manager.
         * Only the first such plugin is returned. If there isn't one then
         * an ArrayIndexOutOfBoundsException is thrown.
         */
        public static ILeaderSupport one(IPluginManager pm)
        {
            return ((ILeaderSupport)pm.lookup(ILeaderSupport.class)[0]);
        }
    }
}
```

Invoking the `enableWarmStart()` method means we set the parameter given to the `warmStart()` method in the `IManagerLifeCycle` plugin. This is application specific, and should be anything which helps the application recover after a complete application failure. Similarly for the `setJoinArgument()` method.

The following example is adapted from the skeleton application. It shows how an `IManagerLifeCycle` plugin is written to make use of

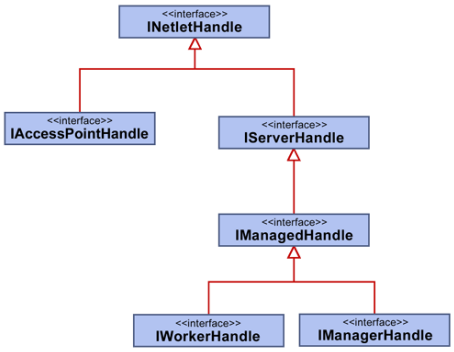


Figure 5.1: The handle interfaces to access remote Beatrix netlets.

the `Manager`'s `ILeaderSupport` plugin, using a leadership change as a trigger to ensure there are the right number of managers running:

```
public class LifeCyclePlugin extends AbstractPlugin
    implements IManagerLifeCycle, IPlugin
{
    // ... much work omitted

    public void changeLeaderMode(boolean you_are_leader)
    {
        if (you_are_leader)
        {
            ILeaderSupport ls
                = (ILeaderSupport)pluginManager().lookup(ILeaderSupport.class)[0];
            ls.setRedundancy(2,4);

        }
    }
}
```

5.4 Being part of the peer group

This section covers general facilities for workers and managers. It comes under the label of "peer group" which is just Beatrix's way of talking about all its netlets in the same application. Communication is an important part of this, but it includes other general ideas which can't be classified elsewhere.

5.4.1 Netlet handles

Reference to other Beatrix netlets is via handles, some of which can are returned by methods in the `IPeerSupport` plugin (Section 5.4.2). These handles are classified in Figure 5.1 and are found in package `org.jtrix.project.beatrix.handle`. Their different features are as follows:

INetletHandle Basic facilities applicable to all Beatrix netlets: get its address, it is still alive, etc.

IAccessPointHandle A basic netlet handle with the extra facility to see what service types it's providing. For example, an access point to our skeleton application offers the *skeleton* service type and the *admin* service type.

IServerHandle An extension to `INetletHandle` which adds the facility to create a service connection with that netlet.

IManagedHandle An extension to `IServerHandle` which adds the facility to communicate with that netlet not just as a service provider, but as a managed netlet in its own right. E.g. we can get a status report from it. Only managers and workers are considered managed netlets because only they are under direct control of the application. Access points spring into life after a consumer performs a service bind, which is clearly an event outside the control of the application.

IWorkerHandle An extension to `IManagedHandle` specialised for a worker: we can get its resources and its worker type.

IManagerHandle An extension to `IManagedHandle` specialised for a manager: we can ask if it's the leader.

5.4.2 Peer support

The `IPeerSupport` plugin, provided by both the `Worker` and `Manager`, allows a netlet to link up to its peers in the application and perform other sundry tasks:

```
public interface IPeerSupport
{
    /**
     *  Provides a server handle for the current leader netlet.  Waits
     *  up to <code>timeout</code> milliseconds for a leader to be
     *  elected if there is currently no leader.  Returns null if no
     *  leader exists when the timeout expires.
     * @param timeout   In milliseconds.
     */
    public IServerHandle locateLeader(long timeout);

    /**
     *  Returns handles for the access points being serviced by this
     *  netlet.
     */
    public IAccessPointHandle[] listAccessPoints();

    /**
     *  List all netlets which can support the given service type.
     *  This is the set of managers or workers who have indicated
     *  service eligibility for the type, for example, by plugging in
     *  the right service provider.
     */
    public IServerHandle[] listServers(String service_type);

    /**
     *  Make this netlet eligible (or not) to serve the specified
```

```
     *  service.  New access points for the service may then connect
     *  to this netlet. It is likely this is not used directly, as it is
     *  called by any IServiceManager when an IServiceProvider or
     *  ISimpleServiceProvider plugin is added or removed.
     * @param service_type       Name of service.
     * @param willing_to_handle  Whether this netlet can handle the service.
     */
    public void setServiceEligibility(String service_type,
                                      boolean willing_to_handle);

    /**
     *  Set simple string-based status output for this netlet.
     *  It can be read from IManagedHandle.
     */
    public void setSimpleStatus(String status);

    /**
     *  Set complex property-based output for this netlet.
     *  It can be read from IManagedHandle.
     */
    public void setComplexStatus(PropertySet status);

    /**
     *  Return this netlet's label, as assigned by its creator at
     *  creation time.
     */
    public String getLabel();

    /** Retrieve Beatrix-provided console, if we want to merge in our own
     * commands.
     */
    public IConsoleFacet getStandardConsole();

    /**
     *  Creates a client session to be given to other services
     *  upon binding.
     * @param critical If true then this is a critical
     *                 service and termination of the service
     *                 results in the netlet terminating.
     */
    public IService createClientSession(boolean critical);

    /**
     *  Create a service session which implements the "standard"
     *  facets supported by the framework.  This can be used as a tool
     *  by normal user sessions.  The supplied flag indicates whether
     *  standard administrative mode facets should be included.
     * @param session  Reciprocating service session.
     * @param admin    Flag to indicate whether admin facets should be
     *                 included in the returned session.
     * @return    Service session incorporating all the standard
     *            facets, possibly including the admin facets.
     */
    public IService createStandardServiceSession(IService session, boolean admin);

    /** Ask the node for this netlet be terminated.
     */
    public void requestTermination();

    /** Get a reference to the node on which this netlet is running.
     */
    public INode getLocalNode();

    /** Check whether a warrant belongs to this application and is signed.
     *  @param warrant  Warrant to be checked
     *  @param require_sid  Whether warrant needs to have a service ID
     *      (and signature).
     */
    public boolean verifyOwnWarrant(Warrant warrant, boolean require_sid);

    /** Access to the applications name, public key, certificates.
     */
    public ApplicationCredentials getApplicationCredentials();
```

```
    /** Create a handle to another netlet in this application
     * @param epa the end point address to the other netlet.
     */
    public IServerHandle createServerHandle(EndPointAddress epa);

    /** Utility class helping lookup.
     */
    public static class Get
    {
        /** Get the only instance of this plugin in the given manager.
         * Only the first such plugin is returned. If there isn't one then
         * an ArrayIndexOutOfBoundsException is thrown.
         */
        public static IPeerSupport one(IPluginManager pm)
        {
            return ((IPeerSupport)pm.lookup(IPeerSupport.class)[0]);
        }

    }
}
```

Setting netlet's simple or complex status is an easy way for it make information available to its peers. Once set through this plugin, it can be read by any other netlet which has a handle to it: an `IManagedHandle` interface.

Also `createClientSession()` is worth pointing out as a handy tool. It creates a simple service session with no facets which can be used as a reciprocating consumer service session when binding a service. Its boolean argument allows us to specify if, when the remote service terminates, this netlet should also terminate. In the future `createClientSession()` may automatically add facets to handle financial transactions, making it easier to pay for services.

See Section 7.3 for more on the `Property` and `Oid` classes.

See Section 6.4 for more on `IConsoleFacet`.

5.4.3 Communicating via internal facets

An internal facet is an interface between two netlets in the same application. If we create a facet (i.e. anything which implements the `IRemote` marker interface) then this allows us to wrap it as an internal facet plugin:

```
public interface IInternalFacetProvider
{
    /**
     * Fetch a facet.
     *
     * @param facet   Facet that needs to be returned
     * @param arg     Facet-specific argument.
     * @return    Object which can be cast into the facet type
     *            and used.
     */
    public IRemote getInternalFacet(String facet, Serializable arg)
        throws BeatrixFacetException;

    /** Utility class helping lookup */
    public static class Get
    {
        /** Get the only instance of this plugin in the given manager.
```

```
         * Only the first such plugin is returned. If there isn't one then
         * an ArrayIndexOutOfBoundsException is thrown.
         */
        public static IInternalFacetProvider one(IPluginManager pm)
        {
            return ((IInternalFacetProvider)pm.lookup(IInternalFacetProvider.class)[0]);
        }
    }
}
```

For example we might like a diary available to all our netlets, but only implemented on some netlets. If we have a facet `com.mycompany.IDiary` we can write it as a plugin:

```
public class DiaryPlugin implements IPlugin, IInternalFacetProvider
{
    // ...some work omitted

    public IRemote getInternalFacet(String facet, Serializable diary_name)
    {
        return (IDiary)_diary_collection.getDiary((String)diary_name);
    }

    public boolean init(IPluginManager m, Object arg)
    {
        // ...load up _diary_collection
    }

    public void shutdown()
    {
        // ...write diaries to disk
    }
}
```

We choose netlet N1 to carry this implementation, so elsewhere we add the plugin to its plugin manager. This is in the code for N1:

```
    _plugin_manager.add(IDiary.class.getName(), DiaryPlugin.class, null, false);
```

And now any netlet which has a handle to N1 can get this facet. In netlet N2 we can write this:

```
    INetletHandle n1 = //...handle to netlet N1
    IDiary d = (IDiary)n1.getInternalFacet(IDiary.class.getName(), "Bob Jones");
```

Now N2 can use the `IDiary` interface as implemented on N1. For details on netlet handles see the `IPeerSupport` interface (Section 5.4.2).

What happens in N2's `getInternalFacet()` call is this:

1. Our call to `getInternalFacet()` on the netlet handle calls that method on netlet N1—the `Manager`, `Worker` or `AccessPoint` class.
2. This scans its plugin manager for any `IFacetProvider` plugin that was added with the facet name, *com.mycompany.IDiary* (obtained using the `Class.getName()` method). It finds the `DiaryPlugin` class we added. This plugin will be initialised if it wasn't already.

3. It calls that facet provider's `getInternalFacet()` method with the arguments given: *com.mycompany.IDiary* and *Bob Jones*. So it will return the "Bob Jones" diary from *_diary_collection*.

Some points to note:

- It is not necessary, but strongly advised, that facet names should be the interface names of the facets returned. For example, the above facet returns an `IDiary` and is named *com.mycompany.IDiary*. We use the `Class.getName()` method to ensure we don't make a typo.
- While an internal facet is in use it may die. For example, its source netlet might fail. In this case subsequent use of the facet will throw an error: `org.jtrix.project.libjtrix.rpc.RPCConnectionBus.InvocationError`. The client netlet will have to get the facet again. See Section 5.4.6 for more on links failing.

5.4.4 Caching internal facets

The interface `IInternalFacetCache` in the `util` package allows the same thing but caches the facets. This is more efficient than getting an internal facet ordinarily because only the first fetch makes a network connection to grab the facet—after that it's cached.

```
public interface IInternalFacetCache
{
    /** Fetch a facet from a particular netlet.
     * The facet will be pulled from a cache if possible, or
     * else will be retrieved and added to the cache.
     * @param h  The netlet the facet is to come from.
     * @param facet   The facet name to retrieve.
     * @param argument Arbitrary argument to pass to the facet.
     * @param cachable Flag whether the cache is to be used.
     *                 If false, the cache is ignored, both for
     *                 retrieving the facet and storing it. */

    public IRemote getInternalFacet(INetletHandle h, String facet,
                                    Serializable argument, boolean cachable)
        throws BeatrixFacetException;
}
```

This interface is implemented by the `InternalFacetCache` class in the same package. See Section 5.4.2, next, for how to get netlet handles.

Here's how to use it. We add the plugin by class but retrieve it by interface:

```
    // First add the plugin.
    // _pm is this netlet's IPluginManager.

    _pm.add(InternalFacetCache.class);

    // ...Then later on we can use it to, say, get our diary interface.
    // Notice its getInternalFacet signature is different.
    // n1 is a handle to the netlet. The boolean is to say whether
    // to use caching.

    Object[] fc = _pm.lookup(IInternalFacetCache.class);
    IInternalFacetCache c = (IInternalFacetCache)fc[0];
    IDiary d = (IDiary)c.getInternalFacet(n1, IDiary.class.getName(), "Bob Jones", true);
```

5.4.5 ICommunicationServer

In Beatrix any facet implementation may also implement `ICommunicationServer`. This means it is told when the connection to its client is terminated and can deal with it as appropriate. Ordinarily this is something facet providers would only find out through errors, but the `ICommunicationServer` interface allows this to be handled more gracefully. However, see the Section 5.4.6 for more on this.

The `SkeletonFacet` class uses a degenerate example of an `ICommunicationServer` (Section 4.4).

5.4.6 When links fail

In a distributed environment communications are always prone to failure. Here is how Beatrix helps us deal with failed links, and what our applications need to add to this:

- The `ICommunicationServer` interface allows connection failures to be handled, as already mentioned. This is server-side only.
- The default proxying access point netlet recognises a server failure and re-establishes communications with another.
- When we have an `INetletHandle` we can add a connection listener which gets called when our link to that netlet fails. `INetletHandle` is available client-side and server-side—see Figure 5.1.
- If a Beatrix netlet binds a service with a warrant and later the link fails, then Beatrix will try to rebind the warrant automatically.

However, even with this last feature there is usually more we need to do. In particular, Beatrix will not rebind any facets we have already bound as part of the service; using of any of these facets will result in an error thrown (`org.jtrix.base.InternalError`, a subclass of `java.lang.Error`). If we detect a connection failure then we might like to take some appropriate action with the unconnected facets.

As an example, a default access point netlet takes each service facet and wraps it. When it detects a link failure (via an `INetletHandle` listener) it binds a new facet. The client only accesses the wrapper and so does not detect any changes.

5.5 Being a manager

Manager-only functions include keeping the application state up to date and keeping track of workers.

5.5.1 Management state

The management state ("MIB") is a memory-only status readable only by the managers, and writable only by the current leader. `IManagementState`, provided by the `Manager`, gives access to it. See Section 7.3 for more on properties, oids, and the management MIB:

```
public interface IManagementState
{
    /**
     *  Set management state properties.  Only allowed on the
     *  leader.  All elements of PropertySet are set as a
     *  single transaction, and will be propagated to backups
     *  as a transaction.
     * @throws NotLeaderException   If called by a netlet
     *                              that's not the leader.
     */
    public void setProperties(PropertySet properties)
        throws NotLeaderException;

    /**
     *  Set a single management state property.  Only allowed on the
     *  leader.
     * @throws NotLeaderException   If called by a netlet
     *                              that's not the leader.
     */
    public void setProperty(Oid key, Property value)
        throws NotLeaderException;

    /**
     *  Get a property set from the management state.  Transactional,
     *  i.e. properties will be fetched from a consistent state.
     *  Missing properties will be null.
     * @param keys  Array of oids whose properties we want to fetch.
     */
    public PropertySet getProperties(Oid[] keys);

    /**
     *  Get a single property from the management state.
     */
    public Property getProperty(Oid key);

    /**
     * Apply an atomic change to the management MIB.
     * @param  changer  Something whose <code>apply</code> method will
     *                  apply a single change to the MIB.
     * @return    The object returned by the <code>changer</code>.
     * @throws NotLeaderException   If called by a netlet
     *                              that's not the leader.
     */
    public Object applyChange(IStateChanger changer) throws NotLeaderException;

    /** Get a read-only view of management state. Can be done by any manager.
     */
    public IPropertyView getView();

    /** Utility class helping lookup */
    public static class Get
    {
        /** Get the only instance of this plugin in the given manager.
         * Only the first such plugin is returned. If there isn't one then
         * an ArrayIndexOutOfBoundsException is thrown.
         */
        public static IManagementState one(IPluginManager pm)
        {
            return ((IManagementState)pm.lookup(IManagementState.class)[0]);
        }
    }
}
```

Notice use of `NotLeaderException`, which is a checked exception and

therefore needs to be caught. Even if the leader calls one of these methods this exception might be thrown, because leadership could change half-way through the call. The netlet should deal with it appropriately.

One clever way to deal with this is to catch the exception, then rethrow `NoLongerLeaderException`, which is in `org.jtrix.project.beatrix.-common`. This is an unchecked (runtime) exception. By throwing this the unchecked exception will percolate all the way back to the access point, which can pick it up, attempt to rebind to a server (getting the new leader in the process) and trying the original method call again. The default access point does all this.

Also of relevance in `IManagementState` is the `applyChange()` method. It takes an `IStateChanger` interface as an argument:

```
public interface IStateChanger
{
    /** Apply a state change. Any changes applied here will not take effect
     * until the method returns.
     *
     * @param tree    Current management MIB, which this method will alter.
     * @return    Anything. This goes straight out and becomes the return
     *     value of the applyChange method of the IManagementState plugin.
     */
    public Object apply(ITypedDataTree tree);
}
```

The principle here is that an `IStateChanger` makes a single atomic change. Its `apply()` method takes the current MIB as an argument and alters it, perhaps based on its current content. Also, the `Manager` ensures such calls are synchronised, so two simultaneous calls will be sequenced. This means that read-only access to the management MIB is best done via the `IManagementState.getView()` method which avoids unnecessary bottlenecks.

The following example is taken from Jtrix.org's Webtrix servlet engine, and shows how the management state is used to find and create contracts. This is, of course, only a tiny fraction of the whole, but it gives us a taste of properties, `IStateChanger`, and more. Observe that by using an `IStateChanger` Webtrix obtains a "lock" on the MIB, so its next contract ID is unique:

```
public class ContractManager extends AbstractPlugin
    implements IPlugin, IContractManager, IInternalFacetProvider, IContractInfoRequester
{
    IManagementState _state;

// ...Vast amounts of work omitted

    private int getNextContractID(ITypedDataTree tree)
    {
        Oid[] oids = tree.getNames(new Oid(WebtrixOids.CONTRACT, -1, WebtrixOids.NAME));

        int max = -1;
        for (int i=0; i<oids.length; i++)
        {
            int t = oids[i].getName()[1];
            if (max < t)
                max = t;
        }
```

```
        if (max == -1)
            return 0;
        else
            return max+1;
    }

    public synchronized Contract createContract(final X500DN key)
        throws ContractException
    {
        if (findContract(key) != null)
            throw new ContractException("Contract already exists");

        Contract c;
        try
        {
            c = (Contract)_state.applyChange(new IStateChanger()
                {
                    public Object apply(ITypedDataTree tree)
                    {
                        int id = getNextContractID(tree);
                        Contract c = new Contract(key, id);
                        tree.setLeaf(new Oid(WebtrixOids.CONTRACT,
                                             id, WebtrixOids.NAME),
                                     new Property(key));
                        tree.setLeaf(new Oid(WebtrixOids.CONTRACT,
                                             id, WebtrixOids.CN_OBJECT),
                                     new Property(c));
                        return c;
                    }
                });
        }
        catch (NotLeaderException e)
        {
            // Percolate an unchecked exception back to the access point
            throw new NoLongerLeaderException();
        }

        c.setPluginManager(pluginManager());
        return c;
    }

    public Contract findContract(X500DN key)
    {
        ITypedStaticDataView view = _state.getView();
        PropertySet contracts = view.getTypedViewEntries(
            new Oid(WebtrixOids.CONTRACT, -1, WebtrixOids.NAME));

        Oid[] keys = contracts.getKeys();
        for (int i=0; i<keys.length; i++)
        {
            Property p = contracts.getProperty(keys[i]);
            if (p.getType() == Property.X500DN_TYPE && p.toX500DN().equals(key))
            {
                Oid o = new Oid(keys[i].getPrefix(2), WebtrixOids.CN_OBJECT);
                PropertySet contract_view = view.getTypedViewEntries(o);
                p = contract_view.getProperty(o);
                Contract c = (Contract)p.toObject();
                c.setPluginManager(pluginManager());
                return c;
            }
        }

        return null;
    }
}
```

5.5.2 Manager support

The `IManagerSupport` plugin is provided by the `Manager`. It is mainly concerned with locating workers, but also allows the manager to generate warrants. Warrants and other service issues are dealt with in Section 5.8.

```
public interface IManagerSupport
{
    /** Make up a new worker name for the given worker type.
     * @param worker_type  Worker type = netlet subtype.
     * @return New unique worker name or null if it could not create a
     *         worker name.
     */
    public String getNewWorkerName(String worker_type);

    /** Check if a worker name represents a worker of a given type.
     */
    public boolean isWorkerType(String worker_type, String worker_name);

    /**
     *  List all workers of a given type.  Workers are listed
     *  regardless of which manager created them.
     * @param worker_type  Worker type = netlet subtype.
     */
    public IWorkerHandle[] listWorkers(String worker_type);

    /**
     *  List all workers.
     */
    public IWorkerHandle[] listWorkers();

    /**
     *  Get a warrant for the specified service, with embedded warrant data.
     *  If Beatrix has a key pair this warrant will be signed.
     *  This warrant data will be available to createServiceSession calls
     *  (within Beatrix) which passes it straight on to the
     *  createFacetCollection call of IServiceProvider.
     * @param service_type  Service which the warrant will grant access to.
     * @param warrant_data  Arbitrary data which will go into the warrant and
     *                      provides extra information to the service provider.
     */
    public Warrant getWarrant(String service_type, Property warrant_data)
        throws BeatrixException;

    /** Get the private key for the application, so it can sign and encrypt
     * things. As added when creating the Beatrix launch descriptor, and set
     * in the application descriptor.
     */
    public PrivateKey getPrivateKey();

    /** Set the load balancing comparator for a specific service type. This
     ** Set how netlets should load balance for a specific service type. We set
     *  a comparator to order IManagedHandles in a manner specific to the
     *  service type. Here is a (not very useful) Comparator which orders
     *  netlets alphabetically by name:
     *  <p>
     *  <pre>
     *  int compare(Object o1, Object o2)
     *  {
     *      IManagedHandle handle1 = (IManagedHandle)o1;
     *      IManagedHandle handle2 = (IManagedHandle)o2;
     *
     *      return handle1.getLabel().compareTo(handle2.getLabel());
     *  }
     *  </pre>
     *  <p>
     *
     *  The default for all service types will order the servers with
     *  the one with the least number of connections first.
     */
```

```
    public void setAccessServerLoadBalancer(String service_type,
                                            java.util.Comparator comparator);

    /** Utility class helping lookup.
     */
    public static class Get
    {
        /** Get the only instance of this plugin in the given manager.
         * Only the first such plugin is returned. If there isn't one then
         * an ArrayIndexOutOfBoundsException is thrown.
         */
        public static IManagerSupport one(IPluginManager pm)
        {
            return ((IManagerSupport)pm.lookup(IManagerSupport.class)[0]);
        }
    }
}
```

Here is another instance of the inner class method `Get.one()` for easier access to this plugin.

The method `setAccessServerLoadBalancer()` is used in very specialised circumstances. It allows us to implement custom load balancing for those netlets which access points connect to (i.e. the access servers).

5.6 Deploying and managing workers

An `IWorkerPool` plugin makes it easy to deploy and manage workers. We will consider their resources separately—see Section 5.7 for that.

5.6.1 IWorkerPool plugin interface

Since an application is likely to have different kinds of workers an `IWorkerPool` plugin is helpful. Each type of worker has its own `IWorkerPool` plugin.

A worker pool is a monitoring tool for one type of worker. Monitoring can be stopped and started, with workers created or killed if there are too few or too many. Therefore every manager netlet will most likely have a worker pool for each worker type, with only the current leader actively monitoring. When leadership changes the new leader just needs to start monitoring and non-leaders need to stop.

A worker pool is also a plugin manager, so to each pool is added plugins that deal with that type of worker. Each plugin can be sure it will see only events for that worker type.

Here is the `IWorkerPool` interface:

```
public interface IWorkerPool extends IPluginManager
{
    /**
     * Start monitoring worker pool. Worker life cycle events will get
     * passed to this pool's IWorkerLifeCycle plugins.
     * @param start_immediately  If true, a check should be made and
```

```
     *     workers should be started before start returns. If false
     *     a check will be made at the next appropriate time.
     */
    public void start(boolean start_immediately);

    /**
     * Stop monitoring worker pool, and possibly stop workers.
     * Worker life cycle events will no longer get passed to this pool's
     * IWorkerLifeCycle plugins.
     * @param stop_workers  If true, stop workers. If false, they can
     *     continue, but are no longer monitored by this pool.
     */
    public void stop(boolean stop_workers);

    /**
     * Set workers' minimum redundancy.
     * @param min  Minimum number of workers at any one time.
     */
    public void setRedundancyMin(int min);

    /**
     * Get workers' minimum redundancy.
     * @return Minimum number of workers
     */
    public int getRedundancyMin();

    /**
     * Set workers' maximum redundancy.
     * @param max Maximum number of workers at any one time.
     */
    public void setRedundancyMax(int max);

    /**
     * Get workers' maximum redundancy.
     * @return Maximum number of workers
     */
    public int getRedundancyMax();

    /**
     * Specify type label for workers. Different worker pools
     *  need different worker types.
     * @param wtype  Worker label to distinguish this pool's workers
     *               from those of other pools.
     */
    public void setWorkerType(String wtype);

    /**
     * Get worker type.
     * @return  Worker type
     */
    public String getWorkerType();
}
```

5.6.2 WorkerPool plugin implementation

The `WorkerPool` class is one implementation of `IWorkerPool` in which

- workers from each pool will always run on different nodes. This helps redundancy. It uses the `Manager`'s built-in `IDeployment-Control` plugin to do this.
- worker-started and worker-died events are passed on to any of its plugins which implement `IWorkerLifeCycle`, but only when those events come from its own worker types;
- a periodic check ensures the right number of workers are running;

- any `IWorkerResourceRequester` plugins are used to assess workers' resource needs;
- it tries to reuse old resources which were previously created by this worker pool but which are currently unused, perhaps because the previous netlet terminated unexpectedly.

5.6.3 Worker pool example

The following is liberally adapted from the skeleton application's `IManagerLifeCycle` plugin and shows how a manager uses a worker pool. We discuss it after:

```
public class LifeCyclePlugin extends AbstractPlugin implements IManagerLifeCycle, IPlugin
{
    static final String WORKER_POOL = "worker_pool_1";

    IWorkerPool workerPool()
    {
        return (IWorkerPool)pluginManager().lookup(WORKER_POOL, IWorkerPool.class)[0];
    }

    /** Start the plugin. Make sure 2-4 workers are always running. */

    public boolean init(IPluginManager plugin_manager, Object data)
    {
        if (!super.init(plugin_manager, data))
        {
            return false;
        }

        try
        {
            pluginManager().add(WORKER_POOL, WorkerPool.class, null, false);

            IWorkerPool wp = workerPool();
            wp.setRedundancyMin(2);
            wp.setRedundancyMax(4);
            wp.setWorkerType(WORKER_POOL);

            wp.add(InternalFacetCache.class, false);
            wp.add(SkeletonWorkerManager.class, false);

            // The file system locator ensures the worker will have
            // 1024kB of disk space. The socket factory locator ensures
            // the worker will have networking.

            FileSystemLocator.LocatorInfo info
                = new FileSystemLocator.LocatorInfo(1024, 1024, true, 1, "disk", 5);
            wp.add("disk", FileSystemLocator.class, info, false);
            wp.add(SocketFactoryLocator.class, false);

            // ...Other plugin work omitted
        }
        catch(Exception e)
        {
            Debug.exc(this, e, "Initialise failed");
            return false;
        }

        return true;
    }

    public Warrant coldStart(IPropertyCollection args) throws BeatrixException
    {
        changeLeaderMode(true);
        return managerSupport().getWarrant("admin",null);
```

```
    }

    public void changeLeaderMode(boolean you_are_leader)
    {
        IWorkerPool wp = workerPool();

        if (you_are_leader)
        {
            ILeaderSupport ls = (ILeaderSupport)pluginManager()
                .lookup(ILeaderSupport.class)[0];
            ls.setRedundancy(2,4);

            wp.start(false);
        }
        else
        {
            wp.stop(false);
        }
    }

    // ...Lots of other work omitted
}
```

- Although all this work takes place in the `IManagerLifeCycle` plugin this doesn't have to be the case. Worker pools can be dealt with anywhere within a manager.
- Any manager can construct worker pools. In this example all managers set them up, but we ensure only the leader activates them. That means any manager is capable of taking over the running of them, but only one manager (the leader) is actually organising the workers at any one time. This helps avoid confusion.
- The worker pool set up takes place in the `init()` method, so it's called when the manager starts.
- When we add the worker pool plugin we give it a name, *worker_pool_1*. This is entirely for our own benefit, so we can reference it again easily, and in no way relevant to the worker pool itself.
- By setting the redundancy levels we ensure that when the pool does its periodic checks it will kill or create extra netlets as appropriate.
- By setting the worker type we give this pool its own unique label. When a worker-started or worker-died event comes in the worker pool plugin can see if it came from one of its own workers by looking at its label. We have set the label to be the same as the plugin's label but this is not necessary in general.
- We add four plugins to the worker pool.
  - The `SkeletonWorkerManager` plugin implements `IWorkerResourceRequester` and `IWorkerLifeCycle`. Hence it both decides what resources the workers need, and responds to their life cycle events.
  - The `InternalFacetCache` plugin is a tool used by the `SkeletonWorkerManager`.

  - The `FileSystemLocator` implements `IResourceRequester`, so the `WorkerPool` will ask it for workers' resource requirements. It is configured to demand 1024kB of non-redundant disk space.
  - The `SocketFactoryLocator` also implements `IResourceRequester`, so it is also asked for resource requirements. (However, it doesn't seek any in this example; see Section 5.7 for more on resources in general and this example in particular.)

- The `IManagerLifeCycle.coldStart()` method is only called after the plugin is initialised. Hence the worker pool is already set up by the time it is called and when `changeLeaderMode()` is called. The `managerSupport()` method belongs to our superclass, `AbstractPlugin`, and just returns this `Manager` object's `IManagerSupport` plugin. We have omitted the `warmStart()` and `join()` methods.
- The `changeLeaderMode()` method ensures only the current leader's worker pool is actively monitoring the workers. The `start(false)` call means the leader's pool is started, but workers start up in their own time; it does not not force an immediate check of the workers. The `stop(false)` call means the non-leader's pool stops monitoring the workers but they carry on running—for the leader to monitor.

5.6.4 Worker life cycle events

A manager can respond to worker stop and start events by adding an `IWorkerLifeCycle` plugin to the appropriate worker pool.

In the `LifeCyclePlugin` above we add such a plugin—`SkeletonWorkerManager`. This responds to a worker start event by adding plugins remotely (Section 5.9). It ignores worker died events.

If several `IWorkerLifeCycle` plugins are added to a worker pool then they will be called in the order in which they were added.

5.7 Requesting and using resources

We look at simple resource location and usage first. The skeleton application's approach to the file system is an example of this. More customised resource work can be seen in its approach to the socket factory. We look at this kind of approach afterwards, though it is not recommended on a first read-through of this document.

5.7.1 Locating resources the simple way

A worker's resource requirements are selected by the manager before the worker is deployed so an appropriate node can be selected. The worker, when deployed, can then bind and use those resources.

Plugins to a manager's worker pool allows the right resources to be found. Here is how the skeleton application's manager does it. Variable *wp* is the worker pool plugin:

```
FileSystemLocator.LocatorInfo info
    = new FileSystemLocator.LocatorInfo(1024, 1024, true, 1, "disk", 5);
wp.add("disk", FileSystemLocator.class, info, false);
```

The `LocatorInfo` class is an inner class of `FileSystemLocator` and tells it what is needed from the file system. The parameters we use are:

- minimum number of kilobytes needed;
- maximum number of kilobytes needed;
- whether it needs to be writable;
- redundancy level (1 means data will be written in one place only, hence no redundancy, 2 means dual redundancy, etc);
- our chosen label for this particular resource.
- reuse priority (of which more later).

When the `FileSystemLocator` class is added to the plugin manager it must be given the same label as we gave the `LocatorInfo`.

Now all workers in that worker pool will have the required file system space when they start. No other work is needed.

The `SocketFactoryLocator` class has a similar `LocatorInfo` inner class. Here is one example of its use:

```
// Workers need a 3-port socket factory creating ports in the range 6000-6999.
// Ports need to have no redundancy (= 1). Priority set to 10.
// wp is our worker pool.

SocketFactoryLocator.LocatorInfo net_info
    = new SocketFactoryLocator.LocatorInfo(6000, 6999, 3, 1, "net", 10);
wp.add("net", SocketFactoryLocator.class, net_info, false);
```

5.7.2 Using resources

When a worker starts up it can make use of file system and networking resources using the plugin `IWorkerResourceProvider`. This is provided by the `Worker` netlet:

```
public interface IWorkerResourceProvider
{
    /** Bind a resource previously allocated to the provider (worker).
     * @param resource_name  Previously set resource name.
     * @param service   Reciprocating consumer service session. */

    public IService bindResource(String resource_name, IService service)
        throws BeatrixServiceException;

    /** Query which resource names have not been previously used
     * and hence need initialising.
     * @return  Fresh resources, labelled with the previously set names.
     */
    public String[] getFreshResources();

    /** Get startup argument, as set by the createWorker method of
     * of {@link org.jtrix.project.beatrix.handle.INodeHandle}.
     * NB: This is not used by the current Beatrix plugin architecture.
     */
    public Object getArgument();

    /** Specify a network provider for this netlet.
     * @param sf  Socket factory facet which will provide the networking.
     * @throws FacetBindException  If the node cannot bind the facet.
     * @throws ITrampolineFacet.InvalidSocketFactoryException
     *             If the socket factory is somehow inappropriate.
     */
    public void setSocketFactory(ISocketFactory sf)
        throws FacetBindException, ITrampolineFacet.InvalidSocketFactoryException;

    /** Insert new file system.
     * @param  prefix  The point at which we will see this file system.
     *             For example a <code>prefix</code> of "/data" will
     *             mean we can write a file called "/data/index.txt".
     *             (Assuming the file system is writable.)
     * @throws FacetBindException  If the node cannot bind the facet.
     * @throws ITrampolineFacet.InvalidMountPointException
     *             If the specified prefix is somehow inappropriate.
     */
    public void mountFileSystem(String prefix, IFileSystem volume)
        throws FacetBindException, ITrampolineFacet.InvalidMountPointException;

    /** Utility class helping lookup */
    public static class Get
    {
        /** Get the only instance of this plugin in the given manager.
         * Only the first such plugin is returned. If there isn't one then
         * an ArrayIndexOutOfBoundsException is thrown.
         */
        public static IWorkerResourceProvider one(IPluginManager pm)
        {
            return ((IWorkerResourceProvider)pm.lookup(IWorkerResourceProvider.class)[0]);
        }
    }
}
```

Here is an example:

```
import org.jtrix.base.IService;
import org.jtrix.facet.node.ISocketFactory;
import org.jtrix.facet.node.IFileSystem;

...

IWorkerResourceProvider wrp = (IWorkerResourceProvider)pluginManager()
    .lookup(IWorkerResourceProvider.class)[0];

IService disk = wrp.bindResource("disk", consumer_service_1);
IFileSystem fs = (IFileSystem) disk.bindFacet(IFileSystem.class.getName());
wrp.mountFileSystem("/home", fs);

IService net = wrp.bindResource("net", consumer_service_2);
```

```
ISocketFactory sf = (ISocketFactory)net.bindFacet(ISocketFactory.class.getName());
wrp.setSocketFactory(sf);
```

Once this is done the file system and socket factory can be used transparently.

Things to note are:

- The process of a worker getting a handle on one of its allocated resources is called *binding*. Hence the method called `bindResource()`. This is just like a consumer netlet binding a service with a warrant; it even gets an `IService` in return.
- A worker binds each kind of resource using the same label as the manager originally used.
- *consumer_service_1* and *consumer_service_2* are any reciprocating client services (`IService`), just like a consumer netlet binding a service with a warrant. We could have used `createClientSession(false)` from our `IPeerSupport` plugin.

Some notes on file system use:

- Only files within the */home* hierarchy can be used in this example. We can call our files and directories */home/jim*, */home/ulf/robots.txt*, etc. Adding another file system with another prefix gives us another hierarchy.
- The prefix */home* is entirely for our benefit. It has no bearing on what directories might be called in the underlying operating system.
- The prefix can only specify the top level directory; prefixes of */var/tmp* and */usr/local* are not allowed, while */var* and */usr* are.
- Jtrix sets the directory separator to be / (a single slash), regardless of the underlying operating system.
- The */proc* hierarchy is given to us automatically; we don't have to bind or mount anything to access it, it's just there. This gives process-specific files. At the time of writing it contains two directories: *runtime* contains the Java^{TM} runtime JAR, *rt.jar*; while *codebase* contains the other JARs currently in use by this netlet, each named with the label used to download it.[1]

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

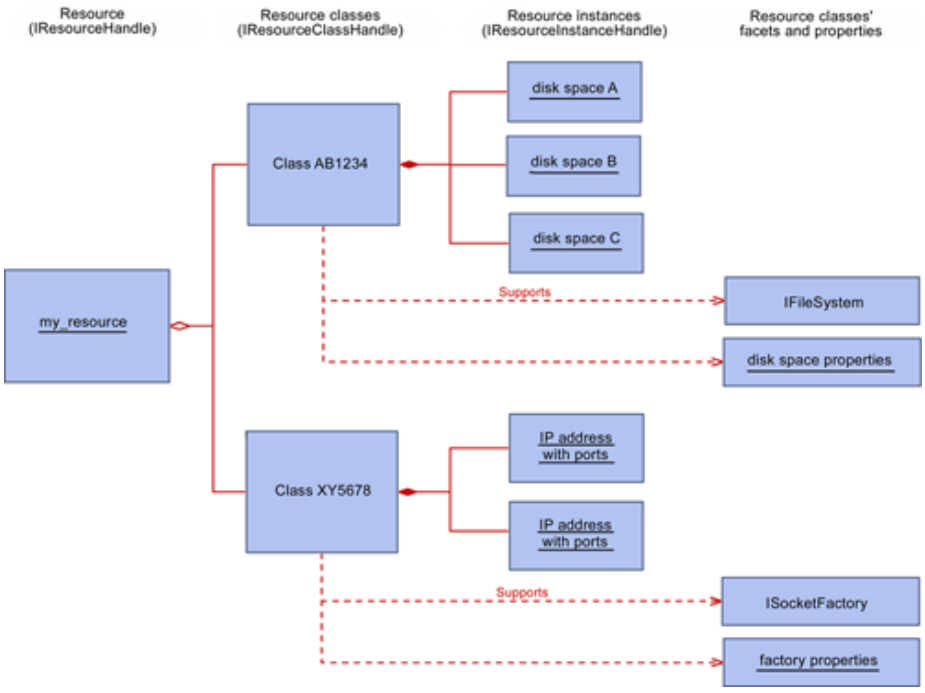


Figure 5.2: Resource concepts and how they relate.

5.7.3 Resource concepts in more detail

This section and the next four deal with deeper resource issues in case we want finer grained allocation. See Figure 5.2. However, be warned: this is a complicted area, and almost everything we need has already been covered. The first-time reader is very strongly advised to skip all this and go straight to Section 5.8. Go now.

Resource instances A *resource instance* is a particular resource earmarked for a particular application. At any one time it may be connected to a particular node. However, just because a resource instance is contracted to a particular application and connected to a node it does not imply that any of the application's netlets are running on that node at that time, let alone using it. It may be waiting for a first netlet to start up and start using it. Or the last netlet might have died, so it will be waiting for another one from the same application to start and pick up where the last one left off.

Examples of resource instances are: 1MB of disk space; 10TB of triply-redundant disk space; a socket factory which can generate sockets listening on port 80; a socket factory which can generate

sixteen sockets on ports higher than 1023; a socket factory sitting behind a redundant load balancer; a CD-ROM containing the names and phone numbers of everyone in the San Francisco Bay Area. In fact, anything can be a resource.

Resource classes Resource instances of the same kind are gathered into *resource classes*. For example, if a hosting service has 2000 little 1MB disk areas then they may all appear in the same resource class.

Each resource class is characterised by two things: resource-specific properties, and what facets it supports.

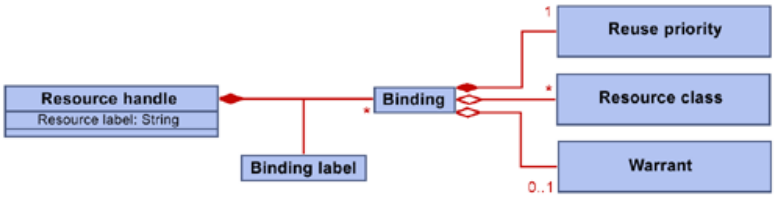
For example, a file system resource class should support the `IFileSystem` facet. But beyond that it may have properties such as kilobyte size and redundancy.

On the other hand, a socket factory resource class should support the `ISocketFactory` facet. But kilobyte size is meaningless to a socket factory. Instead it may have properties describing what port range it supports.

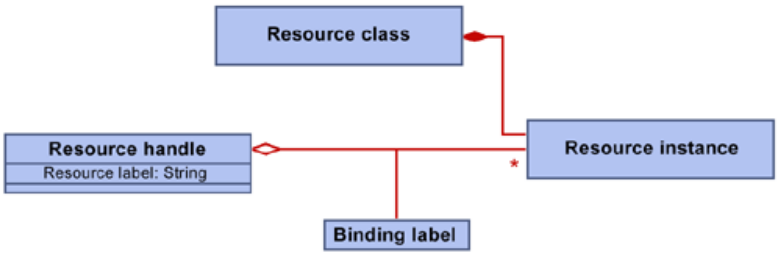
The node's administrator can choose to classify their resource instances how they like. Suppose the administrator has 65535 socket factories, each of which generates a socket on one particular port. There may be one resource class encompassing all these factories, in which case the properties will describe "a socket factory which can generate one socket". Or there may be 65535 resource classes, in which case the properties will describe "a socket factory which can generate one socket on port X" where X is different for each resource class.

Selecting resource classes When a Beatrix manager asks for a resource for an about-to-be-started worker it does not ask for a particular resource instance or even a particular resource class. Instead it asks for all classes which exactly match particular criteria, and then sorts through those by hand. For example, suppose Beatrix wants to find a triply-redundant file system with at least 10MB. It would say "Give me all classes which support `IFileSystem` and which have redundancy property equal to three". When it gets (say) ten resources classes back it examines the properties of each until it finds those with disk space property equal to 10MB or more. (It would be advised to choose the class with no more space than necessary, otherwise it may find it is paying over the odds.) If it doesn't find anything suitable it can try another tactic.

Resource handles Any worker netlet which uses resources will most likely have several requirements: some disk, some sockets. So when a manager has found all the right resource classes (suitable disk space, suitable socket factories, and anything else) it bundles them all up into a *resource handle*. Each resource handle needs a unique *resource label*. A resource handle is also called a *resource* for short.



(a) The resouce handle, before resource instances are allocated



(b) The resource handle, after resource instances are allocated

Figure 5.3: How resource classes and instances and bound to resource handles.

Resource binding The process of bundling resource classes into resource handles is called *binding* resource classes. See Figure 5.3. Since there are several bindings within a resource handle we need to set a *binding label* for each one to distinguish them. In our skeleton example we label the disk binding *disk* and the socket factory binding *net*. Beatrix then takes this resource handle and finds a node with resource instances matching the resource classes, contracts them and starts the worker.

When the worker netlet starts up it can bind these resource instances using the same binding labels. Then it can use them. This is the code in Section 5.7.2 above.

Releasing resources Once we are finished with a resource handle we can *release* it. Then the hosting service can clean and remove all its resource instances and allow other contracts to create new instances in their place. If we don't explicitly release a resource it stays contracted to us. This is regardless of whether or not the netlet using it terminated in a planned way.

Re-use priority It follows from this that if our netlet did terminate prematurely then we would want to start a new one to pick up where it left off, reusing its old resources. Therefore, in binding resource classes into a resource handle we can also specify a *reuse priority*. This says to Beatrix "If I had a resource instance from this class before, try to give me the resource handle with that same resource instance again". This is how we can pick up from where old netlets died prematurely.

For example, if we are running a DNS service then any worker might need a socket factory and some disk space. If a worker died we would want to restart another one. But we would want to prioritise reuse of our old socket factory, because all our clients are expecting us to be listening on that address. We can live without the same disk space, because we can replicate our domain name database from another worker.

Fresh resources If a particular resource instance is not being reused then it is a *fresh* resource. The `IWorkerResourceProvider` can give us a list of the fresh resources. For example, suppose our application was a redundant file storage service and the disk space was a fresh resouce. We would need to bring it up to date with the latest files. However, in practice, even if we did get to reuse previous disk space the files could have changed in between the old netlet dying and the new one starting, so we would still have to synchronise it in our own application-specific way.

Using warrants Finally, when a manager binds resource classes to a resource handle it can also specify a warrant. This allows the node to use an external service for the resources. The worker netlet will not know the difference. If resource classes and a warrant are both specified then Beatrix will use the warrant only if none of the

resource classes could be used; if no resource classes are specified then the warrant will always be used.

So in brief summary... Resource instances are classified into resource classes. A manager netlet specifies what resource classes it wants for a worker, and binds them into a resource handle. For each binding it can specify a reuse priority to reuse old resources from netlets which have died prematurely. It can also specify a warrant to use an external service as if it was that resource instance. Either way it must specify a label. The worker will then use the same labels to bind its allocated resource instances and use them.

5.7.4 Why did I need to know that?

Resources are complicated. And, as we saw in Section 5.7.1, Beatrix's `FileSystemLocator` and `SocketFactoryLocator` handle all that work for us. So why did we need to know all the detail in the last section?

Well, if we're happy with those plugins, then we don't. But if either of them fail to find a particular resource then worker creation fails, and that might not be what we want. If disk isn't available then the worker might still want to run, but omit some of its functionality. Or it might delegate that file system work to another netlet. And if it fails to find a 16-port socket factory it might work happily with two 8-port factories.

Accessing the lower levels gives us greater flexibility. The skeleton application does this with its socket factory.

5.7.5 Locating resources in more detail

The skeleton application's use of the socket factory locator did not use its `LocatorInfo` inner class, although it did make use of the file system locator's one. This is the code from the manager's `LifeCyclePlugin`:

```
wp.add(SkeletonWorkerManager.class, false);

...

FileSystemLocator.LocatorInfo info
    = new FileSystemLocator.LocatorInfo(1024, 1024, true, 1, "disk", 5);
wp.add("disk", FileSystemLocator.class, info, false);

wp.add(SocketFactoryLocator.class, false);
```

Both plugins are added to the worker pool.

Whenever the pool needs to start a new worker it looks up all its `IWorkerResourceRequester` plugins. Then it creates a new resource handle and passes it to each such plugin for them add their own bindings. At the end of that the worker pool tries to create a new worker netlet with the requirements specified by the resource handle.

Since `FileSystemLocator` and `SocketFactoryLocator` both implement the `IWorkerResourceRequester` interface they are asked about their requirements. In the above example the `FileSystemLocator` is given a `LocatorInfo` argument, so, whenever it is called via its `IWorkerResourceRequester` interface, it uses this to add its requirements to the worker pool's resource handle.

However, when the `SocketFactoryLocator` plugin is called via its `IWorkerResourceRequester` interface it finds it wasn't given a `LocatorInfo` argument as its init argument. So it doesn't do anything.

But the `SkeletonWorkerManager`, which was also added to the worker pool, also carries the `IWorkerResourceRequester` interface. So it is also called for its requirements. It responds by looking for a socket factory itself. Here's how:

```
public class SkeletonWorkerManager extends AbstractPlugin
    implements IWorkerLifeCycle, IWorkerResourceRequester
{
    //...Much work omitted

    public void configureResourceHandle(IResourceHandle resource)
            throws IWorkerResourceRequester.ResourceUnavailableException
    {
        ISocketFactoryLocator sfl
            = (ISocketFactoryLocator)pluginManager().lookup(ISocketFactoryLocator.class)[0];
        IResourceClassHandle[] net_classes = sfl.locateByCount(1, 1);

        if(net_classes.length == 0)
        {
            throw new IWorkerResourceRequester.ResourceUnavailableException(
                "No resource classes");
        }

        try
        {
            resource.addBinding("net", 10, net_classes, null);
        }
        catch (Exception e)
        {
            Debug.exc(this,e,"Resource creation failed");
            throw new IWorkerResourceRequester.ResourceUnavailableException(
                "Resource creation failed");
        }
    }
}
```

The `configureResourceHandle()` method is the only method in `IWorkerResourceRequester`. It simply takes the current resource handle and adds its own bindings to it. The implementation here actually uses the `SocketFactoryLocator` plugin to do this: it looks it up by its interface then asks it to identify resource classes which can generate one socket with redundancy level 1. If it finds none it fails (though it could try another tactic). If it finds some then it adds a resource binding: binding label *net*; reuse priority 10 (higher than the priority 5 used for the file system above); any of the resources classes we've just found are suitable; and no warrant for non-local services.

So this code is just what the socket factory locator would do if we had given it an appropriate init argument. But here we have exposed the workings and can see there is opportunity to try other tactics if our first resource grab fails.

5.7.6 How resource priority is used

We have already said that a resource binding with a higher priority (higher number) is a request to prefer re-use of such a resource instance. But how are these preferences actually calculated?

Suppose our application requires resource classes $1, \ldots, n$ with re-use priority P_1 , . . . , P_n. Then a resource handle (i.e. one previously used and still contracted to us, but not currently in use by one of our netlets) gets a score of $W_1P_1+\cdots+W_nP_n$ where each W_i is a weighting as follows:

- If the resource handle's resource instance from class i is still available on that node then $W_i = 2$.
- If the resource handle's resource instance from class i is still available but has to be moved from a different node then $W_i = 1$.
- If a resource instance in class i has to be recreated anew then $W_i = 0$.

Beatrix picks one available resource handle and scores it against each node in the hosting service. The resource handle reused on those for which it has the highest score, and its resource instances get reused where possible. If there are no available resource handles then a completely new one is created, with all fresh resources.

Beatrix does not try to split up old resource handles to create a new one with a possibly higher score, nor does it try any resource handles beyond the first one it finds. But whatever happens we get a resource handle with resource instances from all the resource classes we described, whether they are new instances or not.

For example, suppose our application requires *disk* with priority 8 and *net* with priority 10, and Beatrix has a resource handle potentially for use on nodes N1, N2 and N3. N1 needs a new disk instance, but the net instance is connected to that node. It gets score $(8*0)+(10*2) = 20$. N2 has the old disk space still connected on that node, but the net instance could be moved back from N1. It gets score $(8 * 2) + (10 * 1) = 26$. Node N3 has none of the old disk or net instances, which need to be created anew. It gets score $(8 * 0) + (10 * 0) = 0$. Node N2 wins the resource handle.

5.7.7 Where do resources come from?

After all this discussion of resources, it's worth bring ourselves back to Earth with this rather basic question.

The short answer is: the hosting service. Whoever owns the hosting service chooses the resources when they start it up.

Here is how it works in Jtrix.org's implementation. . . On top of Jtrix.org's node implementation, Nodality, is the program *jnode* which takes a

Nodality node and gives it networking communications. Jnode takes parameters to configure resources. An additional layer, *hospitality*, takes one or more Jnodes, allows them to work as a group and adds a hosting service, meaning that it allows the resources on the Jnodes to be offered to third parties.

5.8 Services

An application will almost certainly want to create services for consumers and then issue warrants to them so they can use the services. It will also want to issue a warrant for the application's administrator.

5.8.1 Creating a service

A `Manager` or `Worker` expects an `IServiceManager` to generate services. Each service is identified by a *service type* (string). Here is how the skeleton manager adds a service:

```
pluginManager().add(ServiceManager.class, false);
IServiceManager sm
    = (IServiceManager) (pluginManager().lookup(IServiceManager.class)[0]);
sm.add("admin", new AdminServiceProvider(), null, false);
```

The `ServiceManager` is an implementation of `IServiceManager` and is a utility that comes with Beatrix. It is itself a plugin manager, and so an `AdminServiceProvider` is added to it under the name *admin*, its service type. The `AdminServiceProvider` is an `ISimpleServiceProvider` plugin, which is one of the things the `ServiceManager` implementation expects. By being added with the name *admin*, any service requests for the *admin* service will be passed by the `ServiceManager` to the `AdminServiceProvider`.

So a `ServiceManager` expects either an `IServiceProvider` or an `ISimpleServiceProvider` to be plugged into it. The `ISimpleServiceProvider` looks like this:

```
public interface ISimpleServiceProvider
{
    /**
     * Create a facet collection for this service label.
     * Called once for each service bind request.
     * @param service_type  Service label
     * @param args  Arbitrary service-specific property argument as contained
     *      in the client's warrant.
     */
    public IFacetCollection createFacetCollection(String service_type, Property args);

    /**
     * See if this service is intended for the service's administrator.
     * If so, Beatrix will add its own admin-specific facets.
     * @return True if this service is an admin service.
     */
    public boolean isAdminProvider();
}
```

The IServiceProvider allows extra information through, so the facet collection can act more intelligently:

```
public interface IServiceProvider
{
    /**
     * Create a facet collection for this service label.
     * Called once for each service bind request.
     * @param session        Session for which this facet collection was created
     * @param service_type   Service label
     * @param args           Arbitrary service-specific property argument
     *     as pulled out of the client's warrant.
     * @param ap_properties  Properties set by the access point
     */
    public IFacetCollection createFacetCollection(IService session,
                                                  String service_type,
                                                  Property args,
                                                  PropertySet ap_properties);

    /**
     * See if this service is intended for the service's administrator.
     * If so, Beatrix will add its own admin-specific facets.
     * @return True if this service is an admin service.
     */
    public boolean isAdminProvider();
}
```

If isAdminProvider() returns true then the ServiceManager will add its own facets—see Section 5.8.2, next.

A service session (IService) is simply a collection of facets which live and die together. In creating a facet collection the class FacetCollection in package org.jtrix.project.libjtrix.netlet is often helpful. In all cases, the ServiceManager may add its own standard facets to those returned by the service provider plugin, as described in the next section.

A word about sequence: by the time the ServiceManager is called the access point netlet will already be running. That means that the access point is there when the ServiceManager calls the service provider plugin implementing either of the above interfaces.

Look back at Section 4.3 to see how the manager's AdminServiceProvider is implemented, and Section 4.4 for the worker's SkeletonServiceProvider.

5.8.2 Facets added by the ServiceManager

After our service provider (i.e. our ISimpleServiceProvider or IServiceProvider) has returned a facet collection the ServiceManager reserves the right to add its own facets before passing the whole lot back to the consumer as a service.

If the service provider says it is an admin service then the ServiceManager will add an IConsoleFacet of "standard" commands. In a manager these commands let us shut down the application, make warrants, list the managers' status, and so on. In a worker there are no

commands in this IConsoleFacet, so it's fairly useless. So a service provider for an admin service is really only useful in a manager.

If the service provider says it is not an admin service then the ServiceManager doesn't add any more facets. However, this could change in a future version.

Either way, the facets that the ServiceManager adds are secondary to ours—in the event of a clashing facet then ours gets used and the ServiceManager's gets ignored. Specifically, if we want to add our own IConsoleFacet in an admin service to replace the standard one then we just return one of our own from the createFacetCollection() method. If we want to return our own IConsoleFacet which adds to the standard one, then Section 6.4.3 explains how to use the JointConsole class.

5.8.3 Creating a warrant

For anyone to use a service a warrant is needed. This is created via the getWarrant() method of the IManagerSupport plugin provided with the Manager—see Section 5.5.2.

Here is how we can return a warrant for a service labelled *admin*:

```
IManagerSupport ms = (IManagerSupport) pluginManager().lookup(IManagerSupport.class)[0];
Warrant w = ms.getWarrant("admin",null);
```

Or if this code is in a subclass of AbstractPlugin:

```
IManagerSupport ms = managerSupport();
Warrant w = ms.getWarrant("admin",null);
```

The null argument can in fact be any Property (Section 7.3) and it gets written into the warrant. Whenever the consumer later uses the warrant to bind the service, that Property is passed back to the createFacetCollection() method of the IServiceProvider or ISimpleServiceProvider.

As a caveat to the above, remember that a service provider is really only used by the ServiceManager implementation of IServiceManager. Strictly speaking, the Property is passed to the ServiceManager which passes it to the service provider. If we are using another implementation of IServiceManager then it can deal with the Property as it wishes; it may not use IServiceProvider or ISimpleServiceProvider.

5.8.4 Some notes on warrants

- Warrants can in general contain any arbitrary data. Beatrix has sanitised its warrants to contain essentially just a service type (string) and the arbitrary service-specific property.

- Warrants contain very specific information about how to connect to the service, and in particular URLs of the bind servers (the SAS servers). Since bind servers can come and go, it is possible for these URLs to get out of date. Therefore as a service provider we might like to allow our consumers to update their warrants—each time the `getWarrant()` method is called the latest bind servers get written in.

- The data held in a warrant can be any `Property`, but what is good practice?

  As an example, a DNS service will issue warrants to allow consumers to manage their own domain name and no others. Each such warrant relates to a contract the consumer has agreed with the service provider. The contract will specify the domain name in question, what rights the consumer has, when the contract should be renewed, and so on.

  A simple approach to the warrant data would be to have it carry a single ID number. The ID refers to a contract in the DNS's database. This is helpful if the contract details are large and complex.

  However, perhaps a better approach in this case is to store all the contract information in the warrant data. This way the warrant is self-contained. The DNS service needs to maintain only a minimal database—it only needs to keep a list of cancelled contracts. Assuming warrant data is signed the service provider does not have to worry about tampering. See Section 7.2 on how to make sure warrants are signed.

5.8.5 Using a service

Meanwhile, here is how a client would use a warrant to bind to our skeleton service (the `SkeletonServiceProvider`, Section 4.4). This is just general Jtrix code, to be used by any consumer who is using our service, and not specific to Beatrix.

```
try
{
    // Assumes _node is an INode interface to our node

    Warrant  my_warrant          = // ...Some warrant to the "skeleton" service
    IService my_consumer_service = // ...Some service. If we're using Beatrix this
                                   // could be generated using IPeerSupport

    IService service = _node.bindService(my_warrant, my_consumer_service);
    String sf_name = ISkeletonFacet.class.getName();
    ISkeletonFacet facet = (ISkeletonFacet)(service.bindFacet(sf_name));

    // Now use this facet
}
catch(ServiceBindException e)
{
    // ...
}
catch(FacetBindException e)
```

```
{
    // ...
}
```

Beatrix plugins can get direct access to the local node using the `IPeerSupport.getLocalNode()` method:

```
    IPeerSupport ps = (IPeerSupport) _plugin_manager.lookup(IPeerSupport.class)[0];
    INode n = ps.getLocalNode();
    IService service = n.bindService(my_warrant, my_consumer_service);

    // Or if this is a subclass of AbstractPlugin...

    INode n = peerSupport().getLocalNode();
    IService service = n.bindService(my_warrant, my_consumer_service);
```

Recall that the `IPeerSupport` plugin also contains a simple `createClientSession()` method to help create reciprocating consumer service sessions.

5.9 Managing plugins remotely

A rather ingenious feature of Beatrix's plugins is remote plugin management.

The plugin interface `IRemotePluginManager`, and its implementation `RemotePluginManager`, both live in package `org.jtrix.project.beatrix.plugins`. The `Worker` netlet provides this via an internal facet. It allows a remote netlet to add, remove and lookup plugins as usual.

The plugin interface `IRemoteServiceManager`, and its implementation `RemoteServiceManager`, do the same for service plugins. However, they are in `org.jtrix.project.beatrix.plugins.util` and are not part of the `Worker` by default.

Here is an example of both in action. The skeleton's manager responds to a worker-started event by adding plugins, turning the `Worker` from a do-little netlet into a worker netlet designed for this application. Among the plugins it adds and uses a `RemoteServiceManager`. All this takes place in the skeleton manager's own `IWorkerLifeCycle` plugin because it's the `workerStarted` method that will be called when any new worker starts:

```
public class SkeletonWorkerManager extends AbstractPlugin
    implements IWorkerLifeCycle, IWorkerResourceRequester
{
    public void workerStarted(IWorkerHandle worker)
    {
        try
        {
            IInternalFacetCache ifc
                = (IInternalFacetCache) pluginManager()
                    .lookup(IInternalFacetCache.class)[0];
```

```
            // Find remote plugin manager and add two plugins

            IRemotePluginManager pm
                = (IRemotePluginManager)ifc.getInternalFacet(
                    worker, IRemotePluginManager.class.getName(), null, true);
            pm.remoteAdd(RemoteServiceManager.class, false);
            pm.remoteAdd(SkeletonWorker.class, true);

            // Remotely add a service provider plugin

            IRemoteServiceManager sm
                = (IRemoteServiceManager)pm.remoteLookup(IRemoteServiceManager.class)[0];
            sm.remoteAdd("skeleton",SkeletonServiceProvider.class, null, false);
        }
        catch(BeatrixPluginException e)
        {
            // Adding a plugin failed
        }
        catch(BeatrixFacetException e)
        {
            // Getting an internal facet failed
        }
    }

    // ...Much work omitted
}
```

Notice that the `SkeletonWorker` plugin is told to "eager initialise"—i.e. initialise immediately. This ensures its `init()` method is called which in this case binds its resources, but in general could do anything to bootstrap that netlet.

5.10 Timing events

Finally, a brief word about the `ITimer` interface and the `TimerPlugin` implementation. These are similar to `java.util.Timer`, but allow tasks to be scheduled and cancelled in groups.

The `WorkerPool` plugin makes use of an `ITimer` plugin to periodically check its workers. It uses the first one it finds, and if it doesn't find any then it adds one itself.

Chapter 6

The launcher console

Beatrix applications can be launched onto remote nodes and controlled from the *launcher* console. Control is by virtue of the warrant returned by the `coldStart()` method of the `IManagerLifeCycle` plugin (Secion 5.2.1). Thus we get a warrant by launching the application. Given this warrant, we can disconnect and reconnect to the application at any time from any console in world.

6.1 Launch/management basics

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

To launch an application we need a warrant for a hosting service (that's where the application will initially live, and whose resources it will consume) and probably for a SAS. The SAS warrant is so that anyone using the application can bind it (and hence its netlets).

However, if the application wants to run its own binding then a SAS warrant isn't necessary. For example, we might be launching our own SAS service. Or we might be happy for all our warrants and netlet descriptors to contain embedded JARs, rather than containing JAR references. The significant downside of this, of course, is that they will all be several megabytes long.

To manage an application we need a warrant for its admin service. This service should provide the `IConsoleFacet` (or `IAdvancedConsoleFacet`[1]), from `org.jtrix.facets1.service.common`. The console then simply passes user commands to the application and returns results. See Section 6.4.1.

[1] We will ignore `IAdvancedConsoleFacet` for now. We pick it up again in Section 6.4.4.

Of course, in the grand scheme of things we might not want to manage an application from a console. We might have our own graphical interface, or perhaps our own other console system. All of these are possible within Jtrix, but the default Beatrix implementation assumes that an `IConsoleFacet` will be at least the first point of admin.

Finally, note that *launcher* isn't part of Beatrix. It is quite independent and can be used to launch and manage any other Jtrix application designed appropriately. The key things are:

1. It launches applications using a *launch descriptor*.
2. It assumes a successful launch will return a warrant.
3. It assumes this warrant is for a service with an `IConsoleFacet`. This facet is used to manage the application.

Jtrix has a tool to generate launch descriptors. See Section 4.9.

6.2 Console commands and variables

For more information on console commands and variables, please see "Start running Jtrix: A practical guide" at `http://www.jtrix.org`.

However, some things are worth noting beyond that. What follows is too detailed for that document:

- Every variable is a `Property` and every application command returns a `Property` array. See Section 7.3 on properties.
- The *connect* command connects to an already-running application using `IConsoleFacet`, so the `IService` returned by the warrant should support that if this is to be successful.
- The *dmesg* command returns standard I/O messages as events. See Section 7.5.1 for more on this.
- The *status* command shows the hosting MIB of the current hosting service (Section 7.3.5).

6.3 The launch process in detail

This section can be skipped first time around; it goes into a lot of detail of work under the hood.

6.3.1 The launch descriptor in detail

Recall the launch descriptor looks something like this:

```
<!DOCTYPE application PUBLIC "-//jtrix.org//TEXT application-0.1//EN"
    "http://www.jtrix.org/dtd/application-0.1.dtd">
<application>
    <jar jar_id='beatrix'>
        <url>file:/usr/lib/jtrix/beatrix.jar</url>
    </jar>
    <jar jar_id='skeleton2'>
        <url>file:/usr/lib/jtrix/skeleton2.jar</url>
    </jar>
    <jar jar_id='parser.jar'>
        <url>file:/usr/lib/jtrix/parser.jar</url>
    </jar>
    <jar jar_id='libjtrix'>
        <url>file:/usr/lib/jtrix/libjtrix.jar</url>
    </jar>
    <jar jar_id='facets1.jar'>
        <url>file:/usr/lib/jtrix/facets1.jar</url>
    </jar>
    <jar jar_id='jaxp.jar'>
        <url>file:/usr/lib/jtrix/jaxp.jar</url>
    </jar>
    <codebase codebase_id='ap'>
        <element jar_id='beatrix'/>
        <element jar_id='skeleton2'/>
        <element jar_id='libjtrix'/>
        <element jar_id='facets1.jar'/>
    </codebase>
    <codebase codebase_id='DEFAULT_FOR_ALL'>
        <element jar_id='skeleton2'/>
        <element jar_id='parser.jar'/>
        <element jar_id='jaxp.jar'/>
        <element jar_id='beatrix'/>
        <element jar_id='libjtrix'/>
        <element jar_id='facets1.jar'/>
    </codebase>
    <boot class='org.jtrix.project.beatrix.plugins.netlets.Manager'
          codebase_id='DEFAULT_FOR_ALL'/>
    <cmdline>
        <argument type="string" name="plugins" required="yes">
            org.jtrix.project.skeleton2.plugins.LifeCyclePlugin </argument>
        <argument type="x500dn" name="x500" required="yes">
            O=jtrix CN=skeleton2</argument>
        <argument type="string" name="name" required="yes">
            Initial Manager</argument>
    </cmdline>
    <config>
        <![CDATA[
            ....
        ]]>
    </config>
</application>
```

The XML elements are as follows:

jar Describes part of the codebase. The `jar_id` gives it a handy label. This label is derived either from looking in the JAR's manifest (see Section 4.8) or from its filename, so it may or may not end with *.jar*. Each JAR should have one or more URLs, each one being a possible location of where the JAR can be downloaded from.

codebase Gathers a number of JARs together under one heading. Will be used to create and run a netlet. We have two codebases in this

example: the default codebase which is used for managers and workers, and one called *ap* which we named specially for access points.

boot Tells the launcher what codebase to use for the launched netlet and which class to use—this is the one that implements INetlet. As usual with Beatrix, this is the Manager netlet.

cmdline Specifies how and what arguments are passed to the IPropertyCollection of the coldStart method. Each argument is a Property and its type must be a supported PropertyType. In our example both arguments' values are specified, but they need not be. Any required element which is not specified must therefore be given on the command line as part of the "run" command.

config Arbitrary data passed into the application as a single lump. Beatrix sets this to be a `BeatrixApplicationDescriptor` object, but a non-Beatrix application can have whatever it likes there.

With the possible exception of the config element, we might want to write or edit one of these by hand.

6.3.2 Launch mechanics in detail

When the console launches an application it creates a netlet descriptor which it runs on the specified hosting service. The JARs it needs for this netlet are either embedded into that descriptor or, if a SAS warrant has been given (using the *sas* command above), uploaded to the SAS and referenced from the netlet descriptor. The JARs certainly cannot simply be read from the local file system where they initially reside since the hosting service on the remote node doesn't have access to this.

Of course, every netlet descriptor also carries a signed parameter bean (delivered as the second argument of the `initialise()` method of `INetlet`). For this, the launcher creates an `ApplicationBean` object, which is simply a carrier for all the information from the launch descriptor XML, and loads it up with all the information in the XML. The `ApplicationBean` is defined in `org.jtrix.facets1.util.launcher`. The newly-launched netlet can then cast its received parameter bean back into an `ApplicationBean`, pull out the relevant data, and cold-start to spread itself out. This is why we can define more than one codebase: the newly-launched netlet may want to deploy several kinds of netlet, each with its own codebase.

Actually, the information from the `ApplicationBean` will necessarily have changed between its original XML (example above) and being delivered to the netlet. First, the JAR locations will have changed. In the example above the launcher reads them from local files which are referenced in the XML. But recall the launch process will upload the JARs to the SAS or embed them in the netlet descriptor; the `ApplicationBean`

references these new JAR locations. Second, the hosting warrant and SAS warrant (if any) given on the console command line are put in the `ApplicationBean`, so the newly-launched netlet can access these, too. Finally, the `cmdline` data is replaced by the actual `Property` values given on the command line and/or their defaults.

All this data, not forgetting the arbitary `config` data, should be enough to enable the newly-launched netlet to distribute itself as an entire application.

6.3.3 Signalling launch success

After starting up, the launcher expects the launched netlet to inform it of its success by providing a warrant for console access. This enables the launching user to administrate their newly-deployed application.

The warrant, of course, is the return value of the `coldStart()` method. Beatrix takes this and uses it as the return value of the of its `INetlet.initialise()` method which in turn gets passed back to the launcher. Then the launcher can bind that warrant and access the `IConsoleFacet`.

6.4 Writing application commands

Giving an application a command line interface is straightforward.

When the launcher connects to an application it looks to see if the service offers an `IConsoleFacet`. If so it binds to it and the application's commands are then available at the console command line.

As a simple example, the `SkeletonServiceProvider` plugin (Section 4.4) offers two facets as part of the "skeleton" service, including the `IConsoleFacet`. This contains a command line command to get the application's message. The other facet is `ISkeletonFacet`, which makes the message available to an application via a method call, `getMessage()`.

6.4.1 IConsoleFacet

```
public interface IConsoleFacet extends IRemote
{
    /** Fetch list of supported commands
     * @return  Array of commands.
     */
    public String[] getCommands();

    /** Fetch command description.
     * @throws CommandException  Thrown if command does not exist.
     */
    public String getCommandDescription(String cmd) throws CommandException;

    /** Execute a command.
     * @param command  The command being executed.
     * @param argument  Any command-specific arguments required to execute.
```

```
     * @throws CommandException  Thrown if command does not exist.
     */
    public Property[] execCommand(String command, Property[] argument)
        throws CommandException;
}
```

When a command is run with *exec* in the launcher its name and arguments are passed straight through to the `execCommand()` method. Its array of return values is what the launcher puts into *$0*, *$1*, *$2*, etc.

6.4.2 Console properties

Notice that `IConsoleFacet` extends `IPropertyCollection`. This means it also supports read-only name/value pairs:

```
public interface IPropertyCollection extends IRemote
{
    /** Fetch all the property names.
     */
    public String[] getPropertyNames();

    /** Query a particular application property.
     * @param name  The property we're querying.
     */
    public Property getProperty(String name) throws PropertyException;
}
```

When a console-oriented application wants to provide access to one of its properties, for example *$sas.port*, it needs to implement `getProperty(String)` accordingly. Any such implementation will probably just return the value from a `HashMap`.

However, a clever implementation will expand on this. An example is Jtrix.org's hosting service which interprets dotted integers and returns the corresponding `Property` from the hosting MIB. Thus if we know a particular oid carries a warrant then we can execute this on a console:

```
connect myapp $hosting.30.0.6.2.243.1
```

6.4.3 Even easier consoles

As if implementing these methods wasn't simple enough, a three extra tools exist to make it even easier.

The first is the `JointConsoleFacet` class in `org.jtrix.project.libjtrix.netlet`. Its constructor takes two or more `IConsoleFacet` objects. The result is a new object which combines the commands and properties of all.

```
    IConsoleFacet primary   = // ...  Some console facet
    IConsoleFacet secondary = // ... Some other console facet
    IConsoleFacet both      = new JointConsoleFacet(primary, secondary);

    // Now "both" is a console facet with all the commands and properties
    // of "primary" and "secondary".
```

The second is `ConsoleFacetFactory` in the same package, which generates commands by itself! A new `ConsoleFacetFactory` is constructed by giving an interface which implements `IRemote`. The new factory introspects the named interface and those interfaces it extends. It looks for methods whose parameters and return values are all `Property`-friendly objects and it automatically creates a command for each one. Then to create an `IConsoleFacet` we just give it an object which implements the original interface. We can also give it properties which can be read from the console command line.

```
// ISomeFacet is a facet. We want to turn its methods into console commands.
// "facet_impl" is an actual implementation of ISomeFacet.

Class c            = ISomeFacet.class;
IRemote facet_impl = // ... Some implementation of ISomeFacet

ConsoleFacetFactory fact = new ConsoleFacetFactory(c);
IConsoleFacet cons       = fact.createConsoleFacet(facet_impl);

// Now "cons" is a console facet, and each of its commands is a method in
// ISomeFacet. Each command will be handled by the implementation "facet_impl".
```

The third is the `getStandardConsole()` method of `IPeerSupport` to get Beatrix's standard console. This returns an `IConsoleFacet` of commands standard to that netlet. On a manager these are genuinely useful commands. On a worker there are no commands in this console, so it's not much help. Either way, this is the same `IConsoleFacet` that a `ServiceManager` adds to every admin service provider's facet collection, as described in Section 5.8.2.

```
// Our plugin manager is in variable "pm".  We use the static method
// IPeerSupport.Get.one() to get the one instance of the IPeerSupport
// plugin that we expect in this plugin manager.

IConsoleFacet cons = IPeerSupport.Get.one(pm).getStandardConsole();

// Now "cons" is a the console facet that's supplied with this netlet.
// If we're in a manager netlet this contains admin commands. If we're
// in a worker netlet there are no commands, though this may change in
// future.
```

The `SkeletonServiceProvider` plugin (Section 4.4) uses all of these tools. It first uses a `ConsoleFacetFactory` to turn the `SkeletonFacet` from a class with a `getMessage()` method into an `IConsoleFacet` with a *getMessage* command line command. Then it uses the `JointConsoleFacet` class to add in the commands provided by Beatrix's standard console facet. (Since this takes place on a worker netlet, the standard console facet has no commands in it; the standard console facet for a manager contains all the privileged admin commands.)

6.4.4 IAdvancedConsoleFacet

Once you've spent some time with `IConsoleFacet` you may find it has several shortcomings. These are addressed in `IAdvancedConsoleFacet` which does everything that `IConsoleFacet` does, plus a bit more.

Before we go into the details of `IAdvancedConsoleFacet` we should own up to a small fib. Previously we said that when the launcher connects to a service it looks for an `IConsoleFacet` and uses that as its console interface. Actually, this isn't strictly true. What it really does is look for an `IAdvancedConsoleFacet`, and if it that's not there looks for an `IConsoleFacet` instead, and if it find that then it wraps it up as its more advanced sibling and uses that.

Now, here's what `IAdvancedConsoleFacet` gives us extra:

- Overloaded commands. Like object-oriented method overloading, it allows us to have the same command invoked with different kinds of properties.
- I/O streams. It allows a command to read input streams and return output streams. Very useful if you want to upload a large file from the launcher, or get a running display of output.
- Custom property types. No longer are we restricted to the launcher's `Property` object interpretation such as *{long:1024}*. Now we can make up our own application-specific properties such as *{my_list:one,two,three}* and use these on the launcher command line. This is achieved by the `IAdvancedConsoleFacet` providing a decoder which converts the command line string (in this case *one,two,three*) into `Property` object (in this case perhaps a `String` in a particular format or just an `Object`).

Here is the interface in full:

```
public interface IAdvancedConsoleFacet extends IRemote
{
    /** When there is a problem decoding an application-specific property.
     */
    public class DecodeException extends Exception
    {
        public DecodeException(String msg)
        {
            super(msg);
        }
        public DecodeException() {}
    }

    /**
     *  Fetch the list of supported commands.
     */
    public CommandDescription[] getCommands();

    /** Execute a command.
     * @param command  The command name.
     * @param arguments  All the Property arguments needed to invoke the
     *     command.
     * @param stdin  Some input stream from which the command might like
     *     to read bytes. This parameter must not be null; at least create
     *     an IInputStream with no data.
     * @param stdout  Some output stream though which the command can return
     *     bytes. This parameter must not be <tt>null</tt> although
     *     it is of course allowed to throw the bytes away if it wishes.
     * @return  An array of return values from the command.
     * @throws CommandException  If anything goes wrong executing the
     *     command.
     */
```

```
    public Property[] execCommand(CommandDescription command, Property[] arguments,
        IInputStream stdin, IOutputStream stdout) throws CommandException;

    /** Decode an application-specific property from a string.
     * See this interface's description for an example of this.
     * @param type  The application-specific type given on the launcher
     *     (or other) command line.
     * @param encoded  The string sequence given on the command line, which
     *     this interface needs to interpret.
     * @return  The string, decoded to become a property.
     * @throws DecodeException  If the encoded string cannot be interpretted.
     */
    public Property decode(String type, String encoded) throws DecodeException;

    /** Get a list of application-specific types which can be decoded
     * with this console interface.
     * @return  Each element of the returned array is a string which could
     *     be the <tt>type</tt> parameter of the <tt>decode()</tt> method
     *     of this interface.
     */
    public String[] getDecoders();
}
```

Finally the advanced console facet has the expected useful tools. Have a look at...

- `AdvancedConsoleFacetFactory` for turning any ordinary facet into an advanced console facet via introspection;
- `JointAdvancedConsoleFacet` for combining two advanced console facets; and
- `WrappedAdvancedConsoleFacet` for turning an `IConsoleFacet` into an `IAdvancedConsoleFacet`.

Chapter 7

Important application aids

Other key concepts essential for writing useful applications. . .

7.1 X.500 DNs

An X.500 distiguished name (DN) is just a series of key/value pairs providing a unique name for a single entity. Such an entity is often called a *principal*. Various string keys are standard. Among others: O is organisation, OU is organisational unit, C is country, CN is canonical name.

X.500 DNs are much used in Jtrix, so we have a class org.jtrix.base.-X500DN to help us.

```
// Make a name using org.jtrix.base.Attribute.  Attribute order is not
// important in an X500DN. For Attribute keys, case is not important,
// but it is for Attribute values.

Attribute a1 = new Attribute("o", "jtrix.org");
Attribute a2 = new Attribute("uid", "jim");
Attribute a3 = new Attribute("cn", "Jim Chapman");

// Create a new X.500 DN, and turn it into a String

X500DN jim = new X500DN(new Attribute[] {a1, a2, a3});
String s1 = jim.toString();

// Now s1 is the following String:
//     O=jtrix.org,UID=jim,CN="Jim Chapman"
//
// Demonstrate conversion from a String.  Spaces, commas and semicolons
// act as Attribute separators, so quote marks needed around Attribute values
// which include these (though there's no harm using them otherwise).

String s2 = "cn=\"Jim Chapman\",o=jtrix.org,uid=\"jim\"";
X500DN jim2 = X500DN.parsePrincipal(s2);
boolean test = jim.equals(jim2);                    // test is true
```

7.2 Warrant security

Given an X.500 DN and a key pair, warrants, descriptors, JARs, and so on, can be signed. This prevents tampering.

7.2.1 Creating a key pair

A new key pair can be generated using *hospitality_keygen.jar* from the hospitality project:

```
java -jar hospitality_keygen.jar <basename>
```

This creates two files: *<basename>.prv* and *<basename>.pub* containing RSA private and public keys respectively. The files' format is just the `getEncoding()` method of the `java.security.Key` interface: PKCS#8 for the private key and X.509 for the public key.

In general Jtrix can handle any encryption standard handled by the underlying VM. Beatrix and the cluster happen to use RSA.

7.2.2 Beatrix and certification

To add certification security to a Beatrix application we need to supply an X.500 DN and an RSA key pair when we generate the launch descriptor. The key pair is optional:

```
% jtrixmaker -type beatrix -outfile skeleton-launcher.xml \
     -x500dn o=jtrix,cn=skeleton2 -jardirs /home/nik/build/bin \
     -jars skeleton2.jar beatrix.jar libjtrix.jar jaxp.jar parser.jar \
         facets1.jar \
     -access ap \
     -codebase ap skeleton2.jar beatrix.jar libjtrix.jar facets1.jar \
     -plugins org.jtrix.project.skeleton2.plugins.LifeCyclePlugin
     -key-pair mykey.pub mykey.prv
%
```

Once a launch descriptor has been created with a key pair all future warrants and netlet descriptors will be signed with it.

7.3 Properties and Oids

To allow generalised and efficient data representation in a structured way, Jtrix uses a `Property` class for fundamental objects and object IDs (oids) to structure them. Various tree stores and views are also available.

7.3.1 The Property class (typed data)

This class is in `org.jtrix.facets1.util.properties`.

The `Property` class supports several types of particular importance to Jtrix. They are: `AuditID`, `BigInteger`, `boolean`, `byte[]`, `EndPointAddress`, `JarURI`, `InetAddress`, `int`, `long`, `NetletDescriptor`, `Object`, `Oid`, `PassiveURL`, `String`, `Warrant`, `X500DN`.

Some sample `Property` work:

```
Property p = new Property((int) 1234);

if ( p.getType().equals(Property.WARRANT) )
{
    // Something has gone horribly wrong!
}
else if ( p.getType().equals(Property.INT) )
{
    // We will get here
}

if ( p.getType() == Property.INT )
{
    // We will also get here. Double-equals works on getType()
}

// This next line will throw a ClassCastException if p is not an int.

int orig = p.toInt();
```

The `getType()` method returns a `Property.PropertyType` object which can be tested with the usual equals method or with `==`.

A `Property` is sometimes referred to as “typed data”.

7.3.2 The Oid class

This class is in `org.jtrix.facets1.util.properties`.

Generic object IDs are made with the `Oid` class, which represents a series of dotted integers: 1, 4.3 and 100.0.0.6.8.20129 are all examples of oids. Also a series of no integers is an `Oid`, and this is additionally represented as `Oid.EMPTY`. A negative integer in an `Oid` name represents a wildcard.

```
// Some ways to construct an Oid

Oid a = new Oid();                // The zero-length Oid
Oid b = Oid.EMPTY;                // Now a.equals(b) --- but !(a == b)
Oid c = new Oid(1);               // A single-digit Oid
Oid d = new Oid(4, 3);            // Represented as 4.3
Oid e = new Oid(100, 101, 102);   // The oid 100.101.102
String e2 = e.toString();         // e2 is now "100.101.102"
Oid f = new Oid(new int{6,7,8});  // Arrays help create arbitrarily long oids
Oid g = new Oid(e, f);            // g is now 100.101.102.6.7.8
Oid h = new Oid(new int{3,2,1}, new int{6,5,4});    // h is now 3.2.1.6.5.4

// A -ve in an Oid represents a wildcard

Oid i = new Oid(100, -1, 102);     // An Oid with a wildcard in it
```

```
boolean j1 = e.matchesPattern(i);  // true
boolean j2 = g.matchesPattern(i);  // false. Must have same length
boolean j3 = g.isPrefixedBy(i);    // true

int k1 = g.getLength();            // k1 is 6
int k2 = a.getLength();            // k2 is 0

int m1 = d.compareTo(f);           // m1 is -1, as 4.3 comes before 6.7.8
int m2 = f.compareTo(h);           // m2 is 1, as 6.7.8 comes after 3.2.1.6.5.4
int m3 = f.compareTo(f);           // m3 is 0, as 4.3 is the same as itself

int[] n = d.getName();             // n is the int array {4,3}.
```

7.3.3 The PropertySet class

The real strength of the `Property` and `Oid` classes begins to appear with the `PropertySet` class. This is an `Oid`-to-`Property` mapping which therefore represents a tree structure. Because each property's object ID is an integer sequence a `PropertySet` can be fairly efficient to manage and transmit over a network. It has been inspired by SNMP.

Here is an abbreviated overview of `PropertySet`:

```
public class PropertySet implements Serializable
{
    /** Create a new, empty, PropertySet.
     */
    public PropertySet()

    /** Create a new PropertySet which is a copy of an existing one.
     */
    public PropertySet(PropertySet p)

    /** Returned Oids have prefix of the given pattern.
     */
    public Oid[] getKeys(Oid pattern)

    /** Returned oids match pattern exactly.
     */
    public Oid[] getMatchingKeys(Oid pattern)

    /** Number of mappings.
     */
    public int getSize()

    /** Set or remove a property.
     * @param value   If null then entry is removed from this PropertySet.
     */
    public void setProperty(Oid key, Property value)

    /** Get the property at the given key.
     *  @return null if property not found.
     */
    public Property getProperty(Oid key)

    /** Get all the keys in this set.
     */
    public Oid[] getKeys()

    // ...Some omissions
}
```

7.3.4 Important related concepts

The following terms refer to specific concepts when we talk about properties and their tree structures.

View A fixed, read-only view of a changable `PropertySet`. Once created, a view does not change, regardless of how the underlying `PropertySet` changes.

Name Refers to one of two things: (a) How up to date a view is, represented as a `BigInteger`. The greater this `BigInteger`, the more up to date it is. (b) The oids of a `PropertySet`.

Delta A change difference between one view of a `PropertySet` and a later one. Thus a delta has a "from" name and a "to" name.

Store Like a view, but deltas can be applied to update it. A store is transactional, so if two updates clash, one is guaranteed to fail rather than corrupting the store.

Tree Like a `PropertySet`, but can be traversed more easily and can have listeners attached to detect changes.

7.3.5 MIBs

`Property` and `Oid` objects are used greatly in "MIBs", or *Management Information Blocks*. These are just very large property trees, and two important ones are the *hosting MIB* and the *management MIB*.

The hosting MIB is used by the hosting service to store all its information about the netlets it hosts: their resource usage, their machines' physical location, main class, and so on. Additionally a netlet has work space within the hosting MIB and has selective access to other parts.

The hosting MIB is not available to most netlets. They use the simple `IHostingFacet` which allows them to bind resources but not access the hosting MIB. (All this is invisible to Beatrix applications.) However, manager netlets get to use the `IHostingControlFacet`, as granted by the hosting service's administrator. `IHostingControlFacet` actually extends `ITypedDataView`, described below. However, we should stress again that this is entirely internal to Beatrix, so you shouldn't be seeking to gain access to it unless you're an advanced Beatrix developer.

The management MIB is a similar affair, and is a memory-only store shared between Beatrix managers. Only the current leader can write properties to the management MIB, and these are propagated to the other managers, so the data does survive a manager dying. The underlying mechanism is a property store, so changing a `PropertySet` is atomic.

To see an example use of the management MIB, properties, views and more go to Section 5.5.1 and look at the sample Webtrix code.

7.3.6 Packages and classes

Once we have these concepts, there are a number of useful packages, classes and interfaces:

org.jtrix.facets1.util.properties

IPropertyStore Interface for a store.

IPropertyView Interface for a fixed, read-only, view which does have a name. (I.e. it knows its up-to-date level.)

ITypedDataView Like `ITypedStaticDataView`, but additionally allows us to check which oids have changed in the underlying `PropertySet`.

ITypedStaticDataView Interface for a fixed, read-only view, which has no concept of knowing its name (i.e. its up-to-date level).

Oid The `Oid` implementation.

Property The `Property` implementation

PropertyDelta An implementation of a delta. It knows its "from" and "to" names, indicating where in the `PropertySet`'s history it fits.

PropertySet The `PropertySet` implementation.

org.jtrix.project.libjtrix.properties

DeltaView Implementation of `IPropertyView`.

PropertyParser Helps convert between strings and `Property` or `PropertyType` objects.

PropertyStore Implementation of `IPropertyStore`.

PropertyTree Implementation of `ITypedDataTree` (see next).

org.jtrix.project.libjtrix.tree

ITypedDataTree Interface for a tree.

ITypedDataTreeListener Listener interface for tree changes.

Also: Various implementations of `ITypedDataTree`, enabling trees which are static, remote, aggregated, cached, updated, and so on, depending on what kind of data/network activity is expected.

7.4 Specialised access points

Beatrix's access points by default are simple proxies. Access points always download in response to a service bind from a consumer with a warrant, and by default that access point will always be the same

proxying netlet, sending instructions back to a managed netlet which has the relevant service provider plugin.

To make a specialised access point we need create the launch descriptor. Recall the command line:

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

The *–access* option may be used any number of times to specify different access point definitions. The service type tells Beatrix (or, rather, SAS with the help of Beatrix) that this access point needs to be delivered when a consumer binds the given service. All the JARs are delivered (this is very inefficient and will change) and the netlet is initialised at the main class, which is the one implementing `INetlet`.

See Appendix C.2 for a concrete example of an application (Webtrix) which uses a specialised access point.

7.5 Debugging and event messages

Some ways to help troubleshoot.

7.5.1 Event messages

Since a netlet runs on a remote node, viewing the standard I/O stream (`System.out`, `System.err`) might seem unlikely.

In fact it is available via the console's *dmesg* command, which displays all the latest standard I/O messages from all the netlets on that hosting service. 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`.

7.5.2 The Debug class

We have already seen the skeleton application using the `Debug` class. This is in package `org.jtrix.project.libjtrix.debug`.

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

```
// Enable debug output, including from with Beatrix. 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, "init(): plugin_manager: ", pluginManager().toString());
Debug.msg(Debug.INFO, this, "Adding ServiceManager.class");

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

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

Only after first setting debugging will we get any debug output, including that within Beatrix itself. It will appear in the node running the application, and includes the internal workings of Beatrix, so can be very helpful. Without this only things explicitly printed to standard output will be seen.

Actually, the `Debug.set()` method can be omitted in our code, because we can set its values on the launcher command line when we *run* the application:

```
launcher> run sk skeleton-launcher.xml debug={long:-1} debug-no={long:0}
```

This sets the debug output mask to maximum, and the debug hide mask to zero. In other words, maximum debugging.

Therefore a good way to debug an application is to:

1. Run our own node with hosting service and SAS.
2. Launch the application into our own node with the command line debug flags set appropriately.
3. Watch the debug output in the node.

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

Appendix A

Things to try yourself

Once you've got the skeleton application running, here are some things to try in increasing order of complexity:

1. Change the name of it.

   Put the application in your own package, rename it, create a new launch descriptor and launch it.

2. Change the number of workers and managers.

   The application starts up with 2–4 managers and 2–4 workers. But if you're only running a one-node cluster then it will fail to create more than one of each since `WorkerPool` tries to start them on different nodes. It will keep trying from time to time and keep failing. Change it to one manager and one worker.

3. Rewrite the `getMessage()` command.

   Currently this is generated with a `ConsoleFactory`. Avoid this, and instead provide a *get-message* command by implementing your own `IConsoleFacet`. Then you can return a `JointConsole` again.

4. Add a *set-worker-redundancy* command.

   Add a command that changes the number of workers at will, so you can take it down from 2–4 to 1 remotely. The command will take effect next time the `WorkerPool` does its periodic check. It should be added to the *admin* service; make sure you don't lose all the commands in Beatrix's own `IConsoleFacet`.

5. Add another worker pool.

   Then amend the *set-worker-redundancy* command to apply to either pool by name.

6. Ensure your warrants are signed

Create the launch descriptor using a key pair. This ensures all the warrants will then be signed. Check this as follows: launch the application with label, say, *sk6*; the variable *$sk6* will then have the application's admin warrant; use the console's *dump* command to write the warrant XML to a file; look at the contents of the file.

Then you can also try the following:

(a) Modify the warrant and try to bind it. You should find it won't bind.

(b) Remove the signature, change the warrant, sign it with a new key, and try to bind this. You should find the binding fails before completion. Of course, this process is more involved. To change the warrant you need to use `org.jtrix.project.-libjtrix.warrant.WarrantBean`. Construct a new instance using the unsigned XML from an `InputStream`; apply the sign method using the new private key and an X.500 name; then write out the new XML using the `writeWarrant()` method.

7. Use a third party service.

   No doubt your re-implementation of *get-message* above simply calls the `getMessage()` method from a `SkeletonFacet` object. Adapt this so it gets the message as if it was from a third party service.

   First, change the *get-message* command so that it takes a warrant as a parameter. Then the command's implementation should bind the warrant and get the `ISkeletonFacet`, just like in Section 5.8.5.

   To test this works, first get a warrant using *get-warrant* from the skeleton application's admin service. This command is part of the standard Beatrix console. Make sure you get a warrant for the *skeleton* service. Next connect to this service and use the *get-message* command.

8. Use the management MIB with a *set-message* command.

   Create a *set-message* command which allows the default message to be changed. Store this string in the management MIB using the `IManagementState` plugin. Make sure each worker is updated with the new message, either by alerting the workers when the message changes via an internal facet. Also ensure that new workers get the latest message when they start.

9. Use the warm start argument.

   Ensure the latest message is stored in the warm start argument. This way, if the message is changed and the service crashes, when it's restored it will still output the most up to date message.

Appendix B

jtrixmaker_help.txt

Appendix C

Example Beatrix applications

Three Beatrix applications and their design guidelines.

C.1 DNS

The DNS service enters into contracts that give the client (contractee) the right to administer a single DNS name, and possibly to create sub-names. Ownership of a name under the contract is conveyed by a contract warrant. From this warrant other, more restrictive, warrants can be derived in order to delegate rights on particular names to DNS-aware third parties (e.g. HTTP server).

C.1.1 Contracts

A consumer's contract is for a single DNS name and carries various optional capabilities: to create sub-names, to create A Records, to create CNAMEs and to create MX records.

C.1.2 Actors

There are three actors for the DNS service

Administrator Runs the DNS service. Manages all contracts and top-level names, controls service deployment, issues contract warrants. Primary access via `IDNSAdminFacet`.

Contractee Owns a domain and can subcontract control of the domain and subdomains by issuing restricted warrants to trusted third parties. Primary access via `IDNSContractFacet`. Every contractee is an. . .

Owner Has control of a domain and subdomain. Primary access via `IDNSDomainFacet`. The HTTP service is one example of an owner.

C.1.3 Netlet roles

Only workers can have resources in Beatrix, and DNS only uses IP addresses. Therefore each worker is DNS server.

All services (i.e. for administrators, contractees and owners) are handled by the managers. Workers do not handle services. Updates to domain information is passed from the managers to the workers.

C.2 Web server

Two Beatrix applications together make the Web server example application: an HTTP service front end and a Tomcat-based servlet engine back end (Webtrix). The HTTP front end is designed so that various back ends can be used—for example, a simple static page provider. The Webtrix back end is just one example.

See the document "Start running Jtrix" on the Jtrix.org Web site for an example of this in action.

The most interesting thing about the Webtrix back end is that it understands that its Tomcat engine needs to be physically close to the HTTP server (a worker netlet) which responds to the HTTP queries. Therefore its access points for this are not simple proxies; they are fully fledged Tomcat engines. More details are presented below. . .

C.2.1 HTTP service

The HTTP service has these actors:

Service administrator Issues warrants for site owners to use the HTTP service.

Site owner Uses an HTTP warrant to first negotiate for the resources required by their back end(s). They can provide the HTTP service with a DNS warrant so it can update their host name to match the (possibly variable) IP address of their Web site. As a distinct process the HTTP site owner will also be a Webtrix site owner (see next) and they will feed WAR files into their Webtrix site. Finally they will give a Webtrix back end warrant to the HTTP service so that it can access those WARs and hence make their site available to the general public.

Back end A back end such as Webtrix uses the HTTP service to bind the requested resources.

The HTTP service has uses for these netlets:

Managers Just store the application state. Also update any DNS services as made accessible by a site owner providing a DNS warrant.

Workers Are the actual HTTP servers. The workers in any one worker pool all have the same kinds of resource available to them. When the HTTP service is given a new back end a manager will look for a worker pool with the right resources, or else will start a new one. Then the worker will bind the back end service. Incoming HTTP requests will get routed to the appropriate back end.

Access points Just the default proxy access points.

C.2.2 Webtrix back end

The Tomcat servlet engine wrapped as a Beatrix application has these actors:

Service administrator Issues warrants for use by a site owner.

Site owner Someone who wishes to run a Web site, which is made up of several WAR files. For each WAR they specify a "context path" which maps a URL path prefix to that WAR. Uploads WAR files into Webtrix, or gives a warrant to a storage system where the WARs are kept. (This latter option is not yet implemented.) Is able to get a back end warrant to give to an HTTP service, so the HTTP service can make the site available.

Front end An HTTP service uses Webtrix to get the WAR files and their context paths, thus making the site available.

Webtrix uses netlets as follows. A key feature is that when an HTTP front end binds to Webtrix the access point netlet it gets is a complete Tomcat servlet engine. Thus the servlet engine is in the same place as the HTTP server.

Manager Stores contracts. Stores what WAR files a contract has and what context paths map to what WARs. Receives uploaded WARs and sends each one to a single random worker.

Worker Uses disk space to store the WAR files redundantly. Since a WAR is sent from a manager to one worker, every so often each worker synchronises with the others with the help of a manager.

Access point Two types: (1) Simple proxy for the service administrator and site owner.

(2) An HTTP front end downloads a more complex access point. This is actually Tomcat wrapped with some Jtrix functionality. It

operates in five steps: (a) Binds a disk resource from the HTTP service. (b) Downloads the relevant WARs from a worker and stores them in the local disk space. (c) Sets up the context paths as set by the site owner. (d) Starts Tomcat, mapping these context paths to the relevant WARs. (e) Tells the front end it AJP13-compliant and gives it its AJP13 address, so the front end can use it as Tomcat normally. The AJP13 networking is made available via the default socket factory which every netlet can access. This is the IP address made available by the –shared option of *jnode*. Even if this IP address is 127.0.0.1, that is fine for this purpose, since the access point and the front end worker netlet binding it are on the same machine.

C.3 SAS

The service advertisement service allows an application to make its netlet descriptors available to consumers. The Beatrix-based implementation also allows JAR files to be downloaded.

C.3.1 Services

Each contract is uniquely labelled with the consumer's X.500 DN and can be created only by the SAS administrator via the ISASAdmin facet.

A service warrant allows the consumer access to the `ISASJarMaintenance` facet for uploading JARs and to the `ISASStaticService` facet which allows them to upload netlet descriptors and unsigned arguments to respond to service bind requests.

Each JAR and service is referenced by a label chosen by the consumer. Once uploaded they can retrieve the list of JAR URLs and bind URLs where the uploads may be found. An uploaded JAR can only be altered by uploading it again, referencing the same label. An uploaded service can be altered by re-uploading the netlet descriptor plus unsigned argument, or just the unsigned argument.

The SAS facets can be found in `org.jtrix.facets1.service.sas`.

C.3.2 Netlet roles

Workers handle JAR requests, replicating the JAR files between them. Workers also handle service bind requests, and cache service entries.

Managers replicate service information, and the current leader manages the JAR replication between workers.