Eclipse Virgo
A Technical Overview

White Paper

Version 1.0
March 2012

Table of Contents

Figures

This document is available as OpenDocument text and PDF in the `white-paper` directory of the following git repository:

```
git://git.eclipse.org/gitroot/virgo/org.eclipse.virgo.documentation.git
```

or at the following location:

http://git.eclipse.org/c/virgo/org.eclipse.virgo.documentation.git/tree/white-paper

Please report any inaccuracies or suggest improvements by raising a bug or posting on the Virgo community forum - see Further Information on page 20.

Introduction

The Eclipse Virgo project provides a modular Java server runtime and is part of the Eclipse Runtime (EclipseRT) umbrella project open source project. Virgo is freely available under the Eclipse Public License.

Unlike traditional Java server runtimes, Virgo was designed from the beginning as a completely modular collection of OSGi bundles running on a standard OSGi framework implementation (Equinox). Virgo was also one of the first runtimes to provide support for applications written as well-defined collections of OSGi bundles.

As well as providing web servers based on popular servlet containers, Virgo has a separate kernel which provides many of Virgo's technical innovations and recently a lightweight "nano kernel" has been factored out of the kernel and is being used to build a lightweight web server which will support the Java EE Web Profile. The nano-based deliverables of Virgo 3.5 should appear in the Eclipse Juno 2012 release train.

So Virgo is a prime example of the "stackless stack".[1] Users have built their own servers on top of the Virgo kernel while other users are happy to use the servers provided by Virgo.

After recapping the background, this paper explains the problems that Virgo sets out to solve, the benefits that Virgo provides, and the core technology inside Virgo. It then summarises the Virgo runtime deliverables and the Eclipse-based and stand-alone application development tooling provided by Virgo. The paper concludes with brief surveys of the component technologies embedded in Virgo, the standards supported by Virgo, and some source of further information.

OSGi Introduction

OSGi is a Java modularity standard with reference implementations and compliance test suites. It is developed by the OSGi Alliance (http://www.osgi.org).

Modules in OSGi are JAR files, known as *bundles*, with manifests which describe the packages which are imported and exported by the bundle. The OSGi framework resolves a collection of bundles at runtime and builds a graph of class loaders that route class loading and resource lookup requests between the various bundle class loaders. In this way, OSGi provides runtime modularity and the problems of a global linear class path which plague Java applications.

OSGi defines a life cycle for its bundles, so it is possible to start, update, and stop a bundle at runtime. This can increase the availability of an application or system since it is possible to update it without bringing down the container.

OSGi also defines a powerful service-oriented programming model. Objects implementing agreed interfaces are published from bundles into a service registry and may then be looked up and used by other bundles. Although it is possible to published and look up services programmatically, OSGi defines two component models which make this particularly easy: Declarative Services and Blueprint. Both these component models are supported by Virgo and described later.

History

The Virgo project started life at the end of 2007 as the SpringSource Application Platform and was soon renamed to SpringSource dm Server. The basic goals of the first version of dm Server, shared by Virgo, were:

- To provide a better OSGi platform
- To simplify migration of Java EE applications to OSGi
- To be modular and extensible

[1] A term coined by RedMonk's James Governor http://www.redmonk.com/jgovernor/2008/02/05/osgi-and-the-rise-of-the-stackless-stack-just-in-time/

dm Server 1.0 was released at the end of 2008. Over the following year more and more people tried it out and a community of developers grew up around it. Based on feedback from this community, version 2.0 was released at the end of 2009 and included a separate kernel.

In January 2010 SpringSource announced that dm Server would be contributed to the Eclipse Foundation as the Virgo project. dm Server went out with a bang by winning the Eclipse award for best RT application.

By June 2010, Virgo was listed in eWeek's 25 Best and Brightest Eclipse Development Projects.

Since then Virgo has shipped 2.1 and 3.0 releases. 3.0 included Jetty support and the snaps framework for modular web applications. At the time of writing in March 2012, version 3.5 is in development and includes p2 provisioning support and new deliverables based on a "nano" subset of the kernel.

Virgo Feature Summary

- Web admin console - deploy and manage artefacts, examine diagnostic dumps, and explore bundle wiring, either in a live system or offline following a resolution failure
- Gogo shell - manage Virgo and deployed artefacts using a command line shell accessed via ssh or telnet
- Plans - define the artefacts that comprise an application, optionally making the application atomic to tie the artefact lifecycles together, and scoped to isolate the application from other applications
- Provisioning - automatically supply the dependencies of an application including bundles, plans, plan archives (PARs), and configurations, from both local and remote repositories
- Web container - supports vanilla WAR files, with all their dependencies in `WEB-INF/lib`, and Web Application Bundles, that import their dependencies via OSGi manifest metadata, via the embedded Tomcat-based reference implementation of the OSGi Web Container specification, configured using the standard Tomcat `server.xml`
- User region - isolates the kernel from user-installed applications and enables admins to focus on application artefacts and their dependencies without seeing those of the kernel as well
- Spring 3 – Virgo 3.0 packages Spring 3.0.5, but can easily be configured to use a different version of Spring
- Hot deployment - deploy artefacts to Virgo by copying them into the `pickup` directory, either in archive or exploded form, as an alternative to deploying via the admin console or shell
- Logging - via embedded LogBack, configured in `config/serviceability.xml`, with a rich set of appenders available out of the box
- JMX management, accessed via the admin console, the shell, the Virgo IDE tooling, or directly via a JMX client such as JConsole.

Figure 1 summarises the key externals of Virgo:

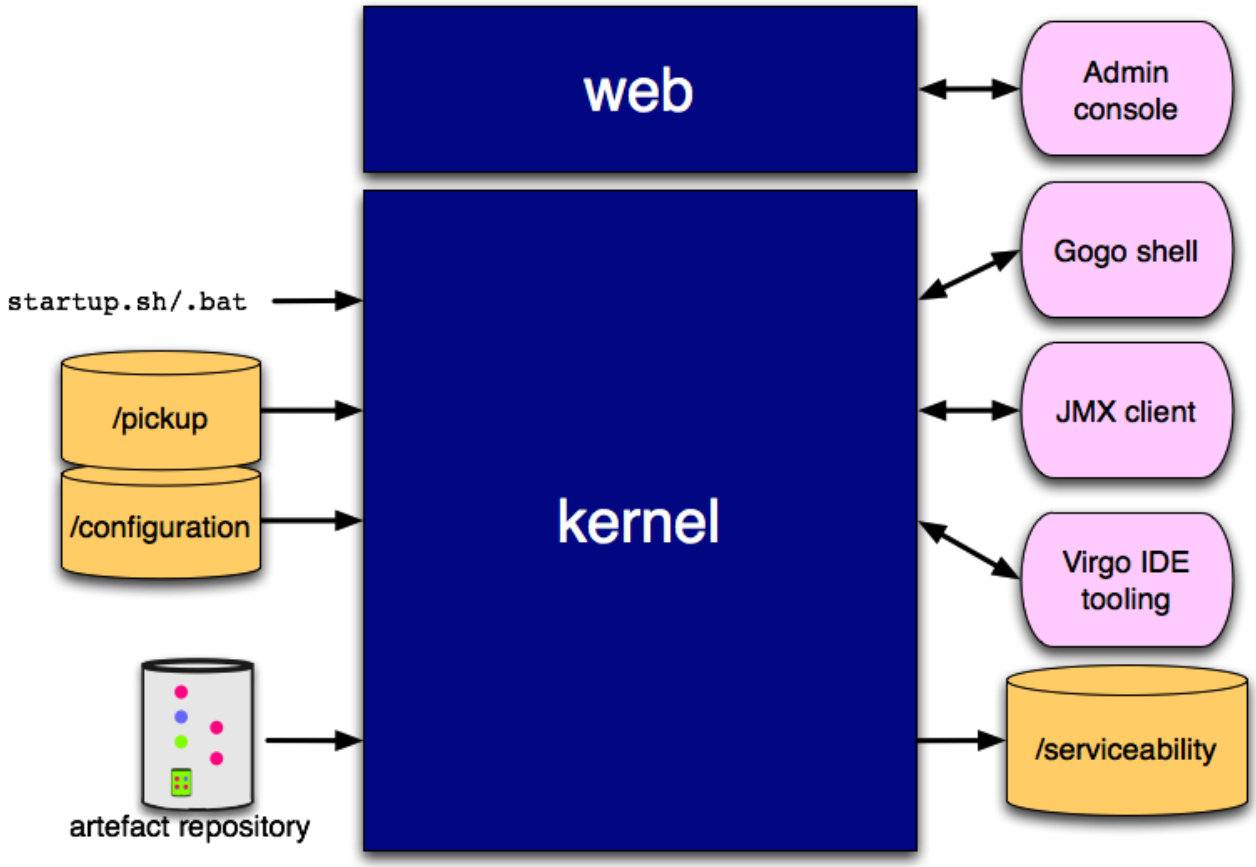

Figure 1: Virgo Externals

Benefits

Virgo is an ideal OSGi server runtime, possibly the ideal OSGi server runtime, from several perspectives.

Standard OSGi

Virgo supports all the standard features of OSGi; encapsulated modules or bundles, a powerful service registry, versioning, class space isolation, sharing of dependencies, dynamic refreshing and updating, and much more.

Bundles All the Way Down

Virgo was designed from the ground up as OSGi bundles so it's extremely modular and extensible.

For example, the core of the runtime is the Virgo kernel and the Virgo Tomcat Server is constructed by configuring a web layer, an admin console, and some other utilities on top of the kernel. Similarly the Virgo Jetty Server is constructed by configuring a collection of Jetty bundles, the admin console, and other utilities on top of the kernel.

The kernel protects itself from interference from such additions, and applications, by running them in a separate *user region*. The kernel and user region are isolated from each other using standard OSGi defined hooks with sharing controlled by a user region configuration file. This separation enables applications to use a different version of the Spring framework to that used by the kernel.

All the major Java EE application servers are now built on top of OSGi, but only Virgo was designed for OSGi and didn't need OSGi to be retrofitted.

Extra Features

Virgo provides a multi-bundle application model to simplify the deployment and management of non-trivial applications. This has helped to inform the forthcoming OSGi Subsystems standard in the OSGi Enterprise R5 specification.[2]

Virgo also provides a repository which can store dependencies such as OSGi bundles which are automatically installed and started when needed. This results in cleaner definitions of applications and a small footprint compared to traditional Java EE servers which pre-load many features just in case an application needs them.

Existing Java Libraries

Virgo enables existing Java libraries to run successfully in an OSGi environment. Admittedly these have to be converted to OSGi bundles first, but then the Virgo kernel supports thread context class loading, load time weaving, classpath scanning, and a number of other features which are commonly used by persistence providers and other popular Java utilities.

Essentially, provides general solutions to the common problems when people attempt to migrate to OSGi.

Container Integration

Virgo integrates OSGi with Spring and a servlet container.

Virgo uses Gemini Blueprint (formerly Spring DM) to wire application contexts to the OSGi service registry. Beans can be published as OSGi services and can consume OSGi services, both with minimal effort.

The embedded form of Tomcat is used as a servlet engine in Virgo Tomcat Server's web support and is configured and managed just like standard Tomcat. Similarly Jetty is the servlet engine in Virgo Jetty Server.

Diagnostics++

Virgo has extensive diagnostics:

- A web admin console enables the state of all artefacts deployed in Virgo to be examined as well as the state of all the bundles in the OSGi framework.
- Virgo provides multiple types of diagnostics: event logging aimed at administrators, diagnostics logging aimed at developers, as well as various types of diagnostic dumps.
- Virgo builds on Logback to support highly configurable and efficient logging.
- When an application is deployed, Virgo first resolves the application in a "side state" and, if resolution fails, no changes are committed to the OSGi framework and the side state is dumped to disk so that it can be analysed using the OSGi state inspector in the web admin console, as shown in Figure 2.
- If a resolution failure occurs, Virgo also analyses the resolver error, including the root causes of any uses constraint violation, and extracts a meaningful message to include in an exception which is then thrown.
- Virgo adds advanced diagnostics to Gemini Blueprint (formerly Spring DM) to track the starting of bundles and their blueprint (application) contexts and issue error messages on failure and warnings when dependencies are delayed.
- Virgo automatically detects deadlocks and generates a thread stack dump.

[2] See “Subsystems” on page 19.

Figure 2: Viewing a State Dump in the Web Admin Console

Advanced Tooling

Virgo tooling support is provided in standard Eclipse via the Virgo IDE update site. This enables a Virgo server to run under the control of the tooling, applications to be deployed, debugged, and updated by the tooling, and package and service dependencies between bundles to be analysed. Figure 3 Shows Virgo running in Eclipse.

Future proof

Virgo is an open source Eclipse project with a liberal license and active participation from multiple vendors, which positions it ideally for the future.

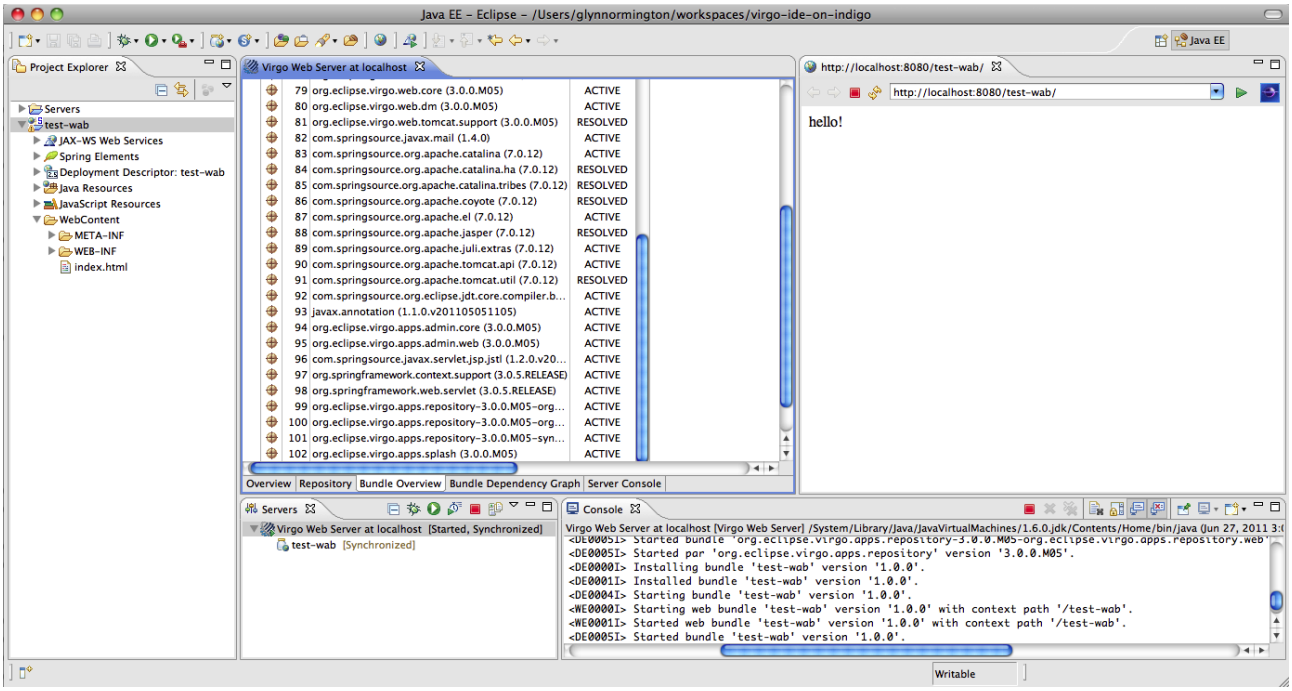

Figure 3: Running Virgo in Eclipse

A Warning

Before discussing Virgo's technology, a word of warning is appropriate. As with any new technology, Virgo is not a “silver bullet”. Virgo is completely modular and is aimed to support modular applications. There is tremendous value in modularising certain kinds of applications, although this often comes with a cost. Two costs should be highlighted to balance the benefits just considered.

The first is the trade-off between the benefits of modularity and the additional conceptual burden of having to deal with modularity at all. It is doubtful whether the additional complexity of modularity can really pay for itself when applied to a very simple application. However, once the size or complexity of an application grows a little, the benefits of modularity start to kick in. When an application reaches moderate size or complexity, the benefits of modularity become clear: the application is easier to maintain, more robust, more flexible, easier to comprehend, test, and debug, and easier to reuse in building other applications.

The second cost is that of refactoring an existing monolithic application into a well-designed collection of modules. The effort involved is often much larger than anticipated and is really only warranted for important applications or those which are anticipated to have a long lifetime and require significant enhancement. It would be hard to justify modularising a fairly stable application or one which has a limited lifespan. However, sometimes the pain of maintaining an application is so significant and the difficulty of making changes without breaking function is so hard that the investment to modularise the application can be justified. Even so, this will not necessarily be an easy or painless process.

The same warnings could have been made of object oriented programming when this started to take over from structured programming in the 1980's. However, software developers are now familiar with object orientation and many have the skills required to design good interfaces and classes. Designing good modules requires the same skills. So there is a sense of inevitability that modular software is the way of the future, at least from an engineering perspective and that the software community will learn to make the appropriate cost/benefit trade-offs.

Technology

Artefacts

Virgo supports the deployment of various types of artefact. Standard artefact types understood by the Virgo deployer are:

1. Bundle – a standard OSGi bundle which is essentially a JAR with special manifest headers;
2. Configuration - a properties file with name of the form `<pid>.properties` which when deployed produces a Configuration Admin dictionary with PID `<pid>`;
3. Plan - an XML file which refers to arbitrary other artefacts in the Virgo repository -- plans may be scoped or unscoped, as described below, and atomic or non-atomic in terms of whether the lifecycle of the plan is tied to the lifecycle of the plan's artefacts;
4. PAR – a Plan ARchive which contains a collection of artefacts and is equivalent to a scoped, atomic plan referencing those artefacts (except the artefacts need not be present in the Virgo repository).

In addition, it is at least theoretically possible to add user-defined artefact types.

Nano

Virgo nano is a small core of function which was factored out of the original Virgo kernel. It provides most of Virgo's diagnostic features as well as p2 provisioning support. Nano can be run on its own to provide a single region framework.

Kernel

The Virgo kernel is the core runtime which may be used on its own or to deploy one or more server types and applications for those server types. The kernel houses the deployment pipeline, described below, as well as support for common artefact types (bundle, configuration, plan, and PAR), regions, scoping, and other core Virgo features.

Regions

Region support was added to enable applications to run with a different version of Spring than that used by the kernel. A minimal set of Spring bundles is installed into the kernel region with very few optional dependencies which keeps the kernel footprint and startup time low. In principle, Spring could be entirely removed from the kernel region if the kernel was modified not to depend on Spring (most of these dependencies are because the kernel uses Gemini Blueprint to publish and find kernel services).

Certain packages and services are imported from the kernel region into the user region and certain services, but no packages, are exported from the user region to the kernel region. This isolates the kernel region from unnecessary interference with the user region. A configuration file controls the importing of packages and services into and the exporting of services out of the user region.

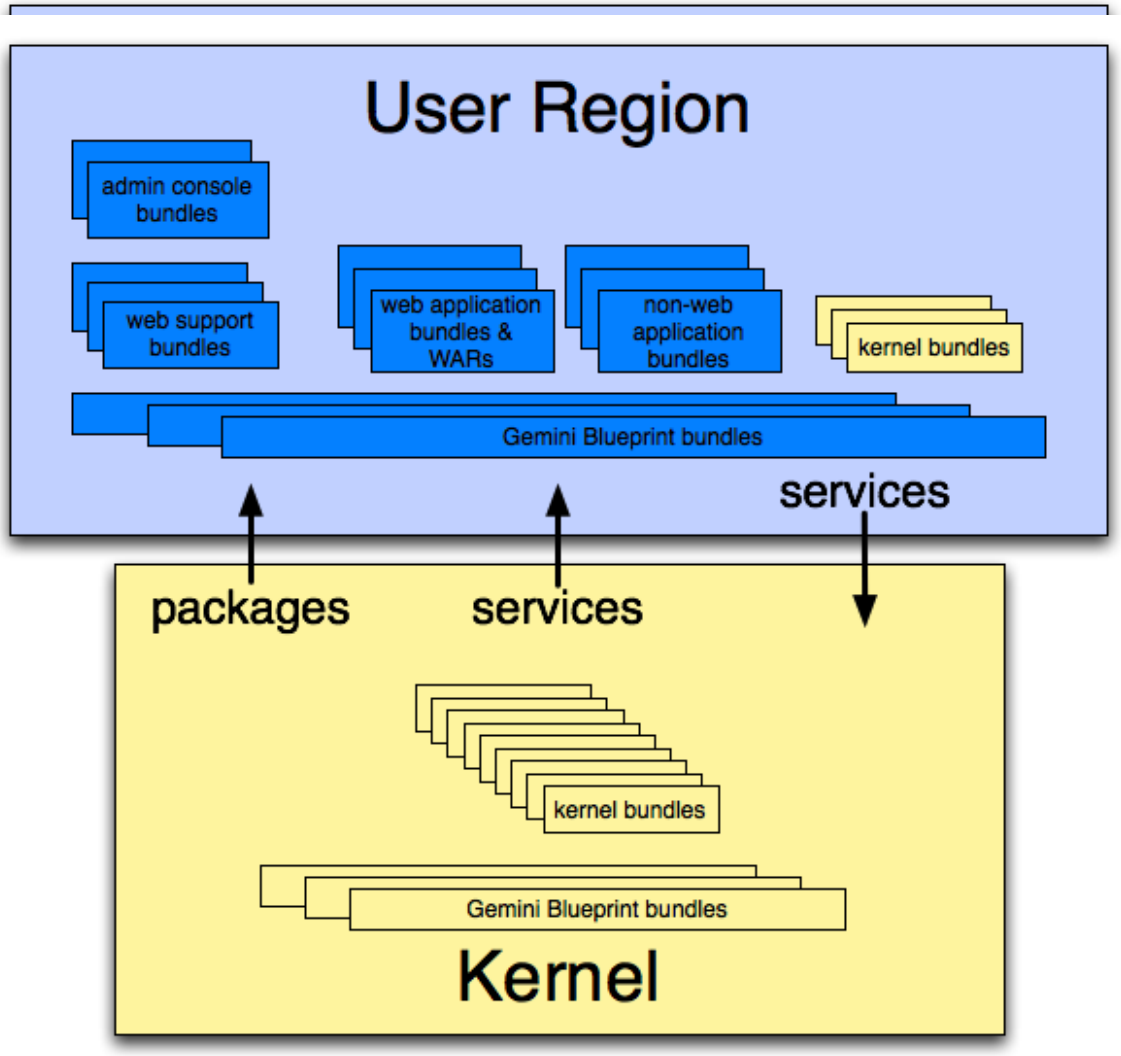

Figure 5: Virgo Regions

So, apart from the basic principle that no packages are exported from the user region to the kernel, there is considerable flexibility for changing the contents of both kernel and user regions and for specifying which packages and services are shared between the regions.

In future, Virgo could be extended to support multiple user regions in order to isolate applications from each other, probably as part of its support for OSGi Subsystems.

Scoping

Virgo adds the concept of scoping to OSGi. The main use case for scoping is where a group of bundles form an application which needs to avoid clashing with other applications and which needs reliable behaviour when it calls third party bundles which use thread context class loading. Clashes can occur because of bundles, packages, or services conflicting in some way.

Metadata Rewriting

Virgo rewrites the metadata of bundles in a scope to prefix the bundle symbolic names with a scope-specific prefix and to add a mandatory matching attribute, with a scope-specific value, to packages exported by bundles in the scope.
Virgo also uses the standard OSGi service registry hooks to limit the visibility of services published by bundles in a scope.
However, a bundle in a scope may access bundles, packages, and services not provided in the scope but which are available outside the scope, that is from unscoped bundles. So a scope acts similarly to a programming language scope such as Java's curly braces:

```
int x;
// b is not visible here
{
    int b;
    // both b and x are visible here
}
```

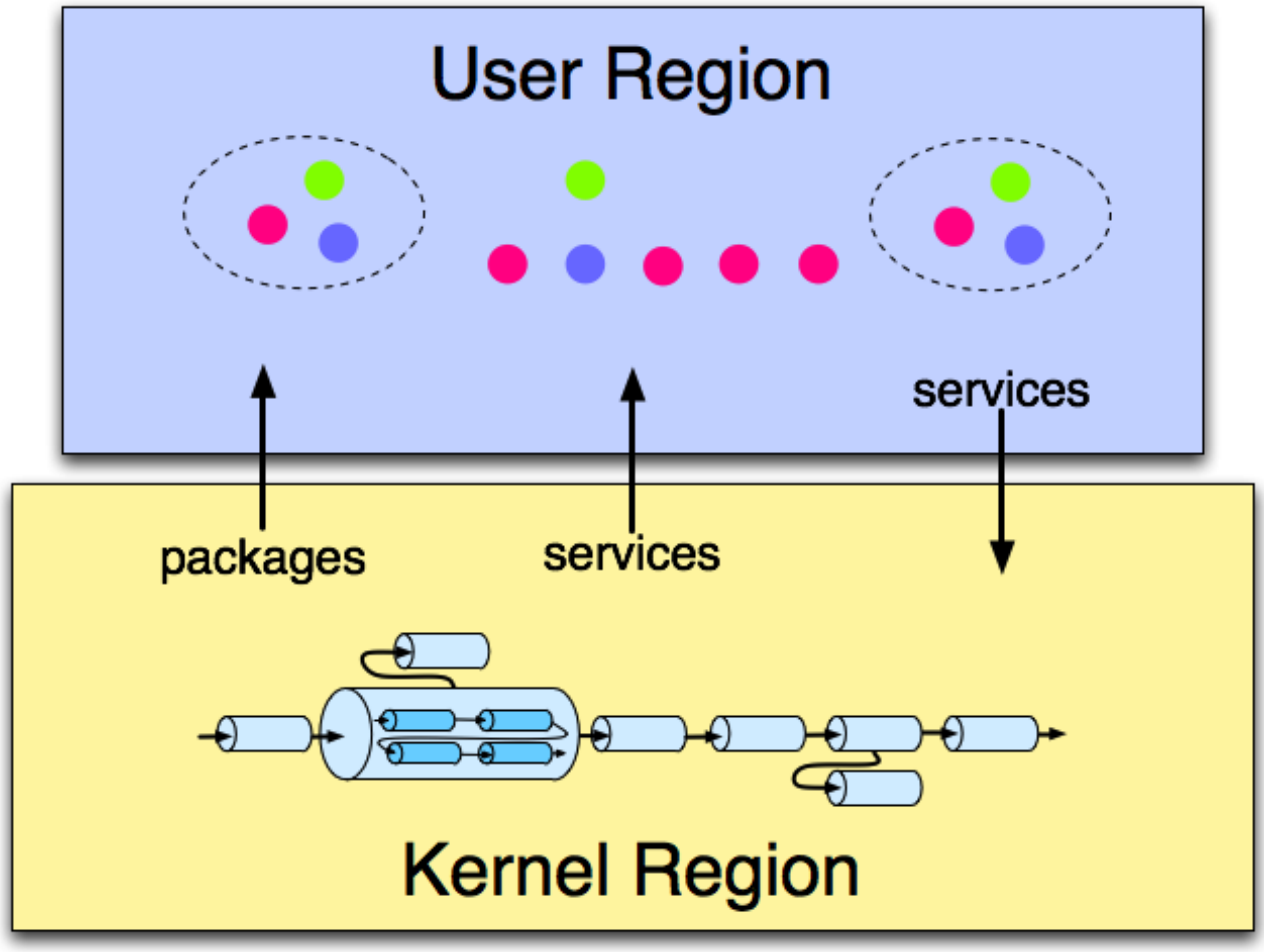

Figure 6: Deployment Pipeline in the Kernel Region

Metadata rewriting is performed using a sophisticated deployment pipeline, described later. This pipeline runs in the kernel region with limited exposure to the User Region, as shown in Figure 6.

Synthetic Context Generation

To ensure reliable thread context class loading when third party bundles are called from a scope, Virgo generates a *synthetic context bundle* in the scope. The class loader of the synthetic context bundle is used as the thread context class loader when bundles in the scope make calls outside the scope. The synthetic bundle imports each of the other bundles in the scope using the Virgo import-bundle header. This is semantically equivalent to importing all the exported packages of the other bundles in the scope. So to make a package of a scoped application available for thread context class loading, it is simply necessary to export the package.

Figure 7 shows an example of a scoped application with two bundles A and B which calls a library bundle L which expects to be able to load application types using the thread context class loader. In particular, L may need to load times from bundle A even when called from bundle B. Using B's class loader as the thread context class loader on the call to L would only solve the problem if B imported the packages exported by A. This approach would obscure the true structure of the application, in which B may not depend on A and make updating bundles have a greater impact than necessary. The synthetic context bundle solves this by providing a thread context class loader which can load all the needed types while not obscuring the structure of the application.

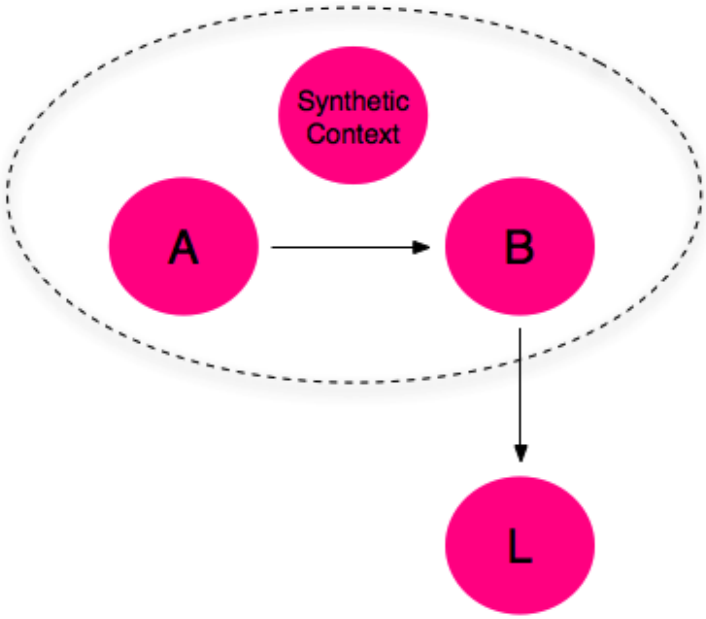

Figure 7: Synthetic Context Example

Example of Scoping

Figure 8 shows a scoped plan referring to two bundles A and B being deployed. The result is a scope containing the bundles A and B as well as the synthetic context bundle. Note that bundles inside the scope can access bundles, such as X, outside the scope. Also, bundles outside the scope, such as Y, cannot access bundles inside the scope.

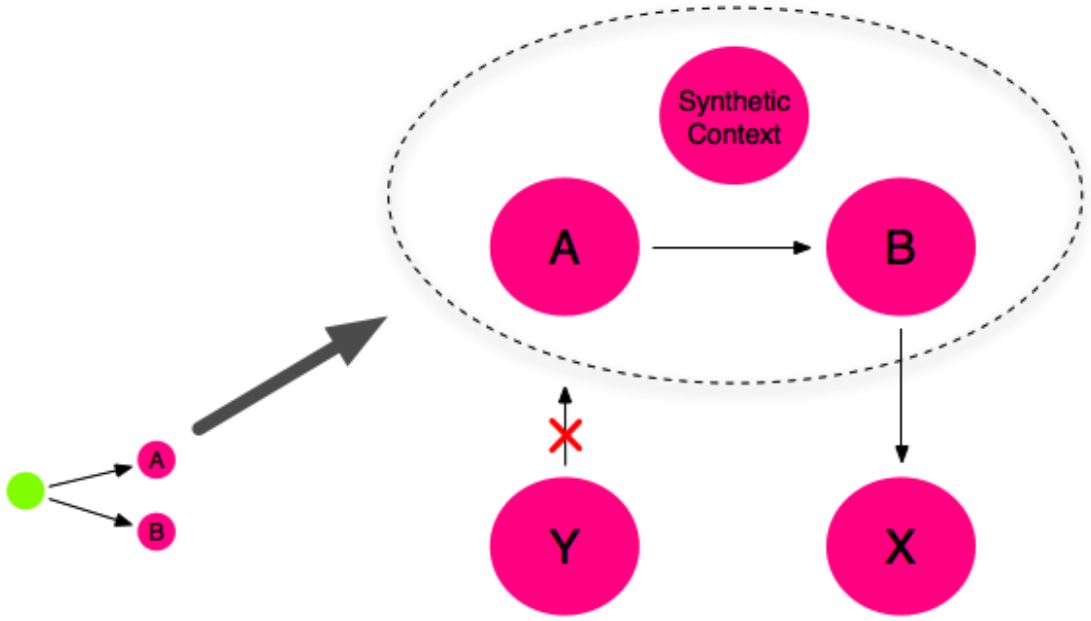

Figure 8: Scoping Example

Deployment Pipeline

Artefacts are deployed into Virgo using a deployment pipeline consisting of several pipeline stages some of which have pipelines nested inside them as shown in Figure 9.

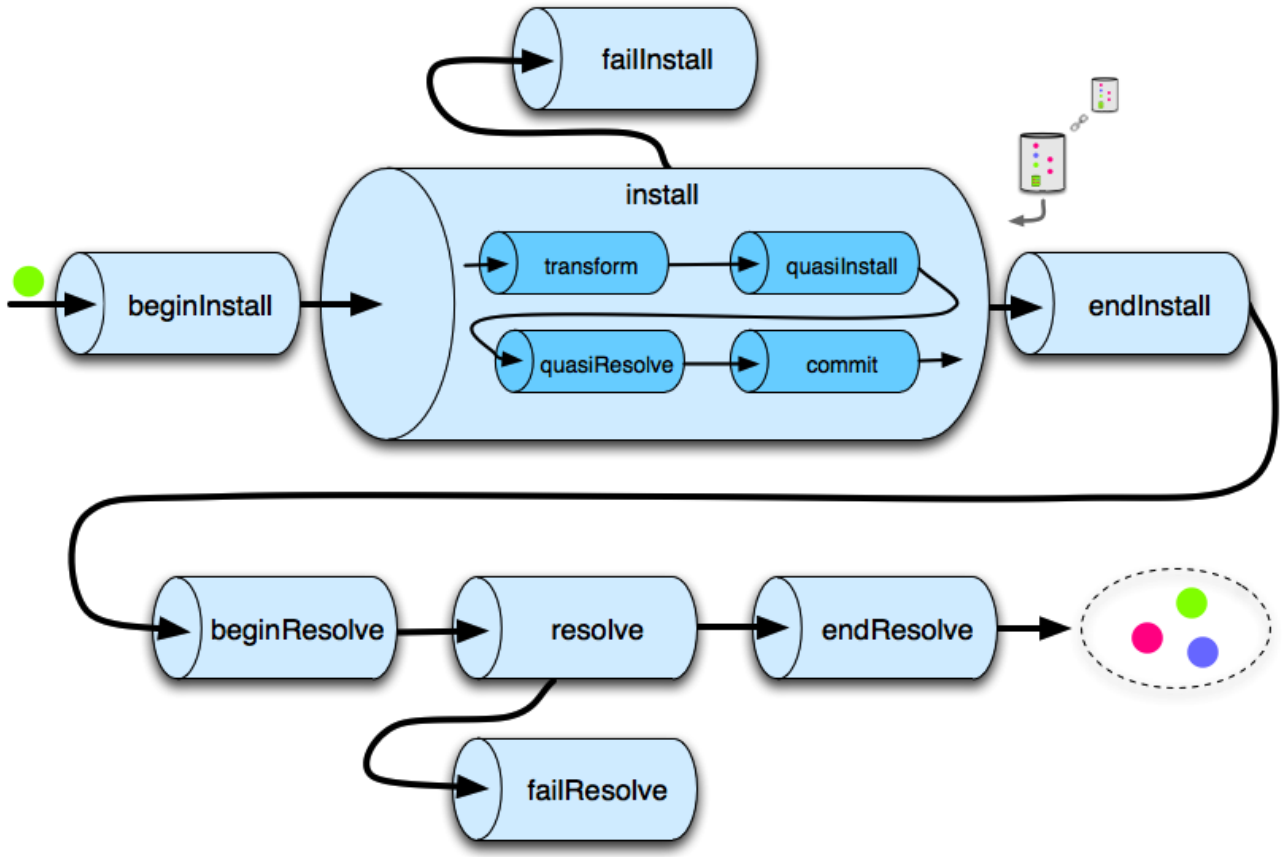

Figure 9: Deployment Pipeline Stages

The pipeline, and each pipeline stage, accepts a tree of install artefacts as input and outputs a possibly modified tree. The deployment pipeline is constructed by the `Plumber` class.

Transformers

Many of the interesting modifications to the tree are performed by the transform stage which uses the whiteboard pattern[3] to drive all services of a Transformer type in order of service ranking. A number of standard Transformer services are defined by the kernel's deployer bundle. Some interesting examples of standard Transformers are:

- `PlanResolver` which takes as input a tree consisting of a single plan node and adds a subtree representing the content of the plan, including any nested plans and their subtrees,
- `ScopingTransformer` which rewrites the metadata of a subtree rooted in a scoped plan and gathers service scoping information for the subtree,
- `SyntheticContextBundleCreatingTransformer` which adds a synthetic context bundle as a child node of a scoped plan, and
- `ImportExpandingTransformer` which converts Virgo-specific headers such as `import-bundle` into standard OSGi `import-package` statements.

Quasi Framework

The quasi framework is an abstraction of the Equinox framework state and is used in auto-provisioning missing dependencies during deployment. The quasiInstall stage installs the bundles in the input tree into an instance of the quasi framework. The quasiResolve stage attempts to resolve these bundles and auto-provision any missing dependencies from the Virgo repository by installing them in the quasi framework instance. The commit stage attempts to install the bundles in the input tree, along with any auto-provisioned bundles, into the OSGi framework.

Exception Handling

There are two approaches to handling exceptions thrown by a pipeline stage. In general, unexpected exceptions are allowed to percolate upward and result in diagnostics and a failed deployment. However, certain expected exceptions, such as failure to resolve the dependencies of the install artefact tree, need to be handled more gracefully. In these cases, a compensating pipeline stage is defined which drives a compensation stage if an exception is thrown. failInstall and failResolve in Figure 9 are examples of compensation stages.

Repositories

Virgo repositories contain artefact URLs and metadata and are indexed by the Cartesian product of artefact type, name, and version. There are three kinds of repository: external, watched, and remote. Repositories are passive in the sense that changes to repository content do not cause artefacts to be deployed into Virgo, refreshed, or undeployed. Repositories support queries which allow sets of artefacts satisfying certain criteria, for example with a version in a given version range, to be determined.

External Repositories

External repositories are created by scanning a directory which contains artefacts, possibly in nested directories. The repository configuration specifies a pattern which says which files should be treated as artefacts. After the repository is created, changes to the directory do not affect the repository content.

The Virgo kernel's default repository configuration specifies an external repository created from the `repository/ext` directory.

[3] In the whiteboard pattern, event consumers register themselves in the service registry under an agreed interface and one or more event producers locate consumers in the service registry and publish events to all the consumers.

Watched Repositories

Watched repositories are created by scanning a directory which contains artefacts but no nested directories. All files in the directory are treated as artefacts. The directory is re-scanned periodically and the interval between re-scans is specified in the repository configuration. Changes detected by re-scanning are reflected in the repository content. Note that changing the content of a watched repository does not cause artefacts to be deployed into Virgo, refreshed, or undeployed.

The Virgo kernel's default repository configuration specifies a watched repository based on the contents of the `repository/usr` directory.

Remote Repositories

A remote repository refers to a repository hosted by a Virgo instance sometimes known as a repository server.

The remote repository is accessed by a Virgo instance sometimes known as a repository client. The repository client is normally a different instance of Virgo to the instance hosting the repository, but it can be the same instance which is handy for testing. The remote repository periodically downloads its content from the hosted repository. The period between downloads may be configured in the repository configuration. The remote repository also caches artefacts which have secure hashes associated with them in the hosted repository. Only bundles currently have secure hashes associated with them. The secure hash is used to determine when a cached artefact is stale and needs to be freshly downloaded.

Repository Chains

The Virgo repository is configured as a chain of external, watched, and remote repositories. The chain is a list which is searched in the configured order. The effect of this search order is that an artefact with a given type, name, and version which appears in more than one repository in the chain is only accessed from the first repository in the chain in which it appears. Abstractly, the repository chain behaves as a single repository, but its content may mutate in quite a different way to the content of an individual external, watched, or remote repository because of the sequential searching of the chain.

Aspects

Virgo is built using compile-time weaving to inject code into Virgo to satisfy various pervasive requirements. AspectJ aspects are used to define *join points* where code is to be injected and *advice* or injected code. Most of the Virgo codebase is instrumented using an Aspect which provides method entry/exit trace logging. This combined with Logback's ability to configure logging at the level of a package or class means that it is easy to configure Virgo so that a particular area of code produces detailed log messages while the rest produces very little. AspectJ makes this easy to maintain as there is no hand-written method entry/exit code to maintain.

Other aspects generate a dump when an unchecked exception is thrown by Virgo and sanitise exceptions thrown under a JMX request so they are suitable for propagating across JMX.

Virgo Runtime Deliverables

Virgo Tomcat Server builds on the kernel and adds Gemini Web, a web integration layer, and

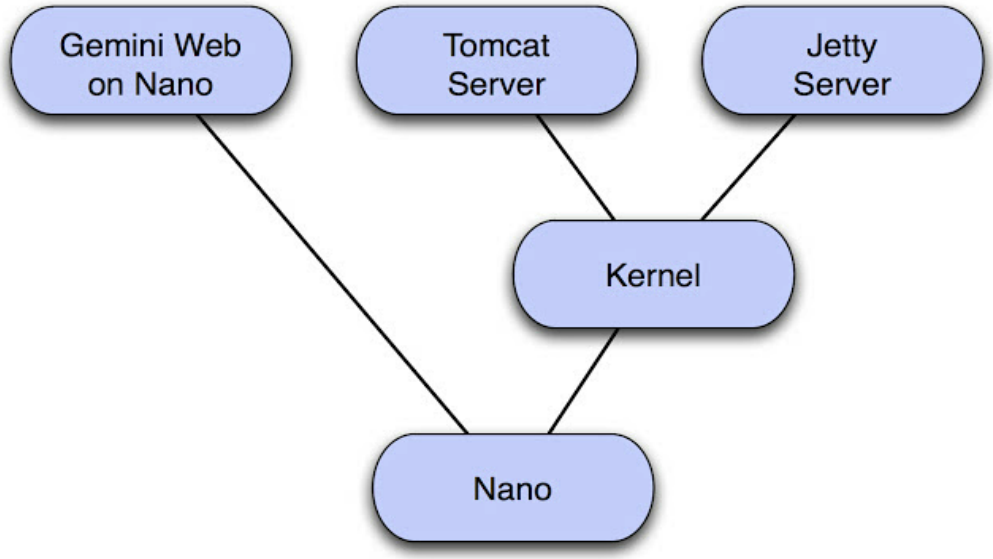

Figure 10: Virgo Deliverables

various supporting bundles. It supports deploying sophisticated multi-bundle web applications. The Virgo snaps framework may also be used to cleanly modularise the web bundles of an application.

Virgo Jetty Server is similar to Virgo Tomcat Server but embeds the Jetty servlet container instead of the embedded Tomcat servlet container. Currently, snaps does not support Virgo Jetty Server.

Virgo kernel supports the core concepts of Virgo but is not biased towards the web, thus enabling other types of server to be created. The kernel can also be used stand-alone as a rich OSGi application platform. A server can easily be constructed by deploying suitable bundles on top of the kernel.

Virgo Nano is a minimal core of the Virgo kernel which is designed to run relatively simple OSGi applications in a single region. Bundles can be provisioned at runtime using p2.

Virgo Nano Full builds on Virgo Nano and embeds Gemini Web to provide Tomcat-based servlet support. Bundles, including Web Application Bundles, can be provisioned at runtime using p2.

All the above deliverables can be initially provisioned using p2.

Virgo IDE Tooling

Virgo provides Eclipse-based tooling which may be installed into a regular Eclipse for Java EE distribution. The Virgo tooling inherits various OSGi standard features from the Eclipse Libra project and then adds Virgo-specific capabilities including the ability to develop and deployer bundles, PARs, and plans to a Virgo instance managed by the tooling.

Virgo Bundlor

Virgo Bundlor is a manifest generation tool which generates a bundle's manifest based on a template provided by the user and bytecode inspection of the bundle's classes. The template defines bundle symbolic name, bundle version and similar headers as well as a policy for exported package

versions and imported package version ranges.

Embedded Technology

Various open source components are reused by Virgo.

Tomcat

An embedded variant of the Apache Tomcat servlet container is embedded in Virgo Tomcat Server and Virgo Nano "full" as part of Gemini Web. Tomcat implements the Java Servlet specification.

Jetty

The Jetty servlet container is embedded in Virgo Jetty Server. Jetty implements the OSGi Web Applications standard and the Java Servlet specification.

Gemini Web

Gemini Web is embedded in Virgo Tomcat Server and Virgo Nano "full" to support the deployment of WAR files and OSGi standard Web Application Bundles (which are approximately WAR files with bundle manifests). Gemini Web is the Reference Implementation of the OSGi Web Applications standard. Figure 11 shows the high level structure of Gemini Web.

Web Application Bundles are deployed into OSGi as normal and an *extender* in Gemini Web listens for bundle start events and registers a corresponding servlet with the Tomcat servlet container. WAR files must be deployed into Gemini Web using a special URL scheme. Gemini Web provides a URL handler which converts such WARs into Web Application Bundles before deploying them into OSGi.

When Gemini Web is reused in Virgo, a web integration layer integrates Gemini Web into the Virgo deployment pipeline so that WARs and Web Application Bundles can be deployed using all the normal Virgo deployment mechanisms: web admin console, pickup directory, Virgo IDE tooling, etc.

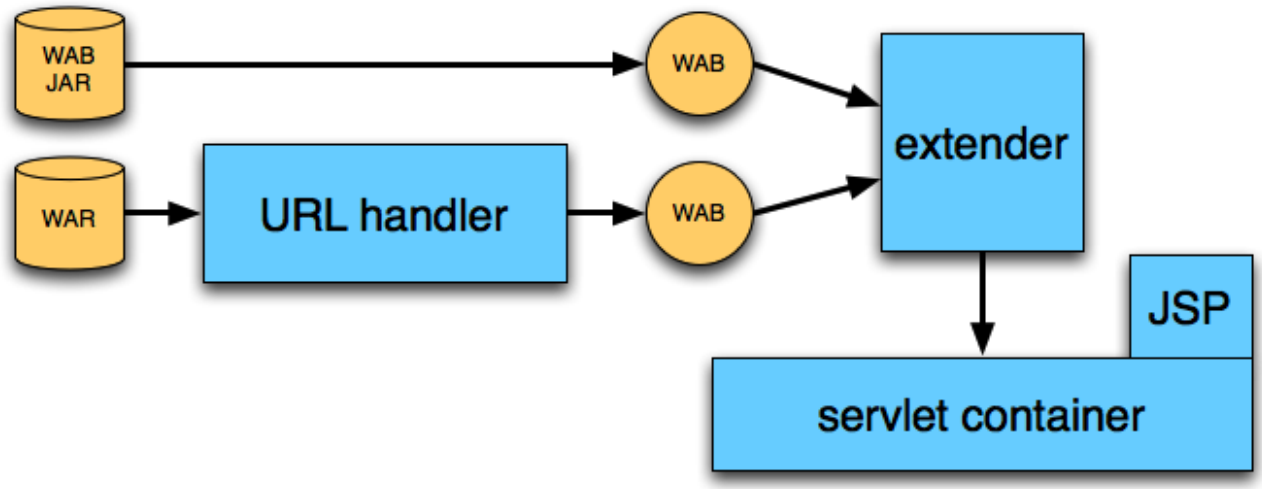

Figure 11: Gemini Web

Gemini Blueprint

Gemini Blueprint allows services and references to services to be declared in XML. Service declarations specify the concrete class which is instantiated to provide the service. Service references are injected into bean declarations to provide a Spring-like dependency injection

programming model. Gemini Blueprint also supports the earlier Spring DM function from which it was derived.

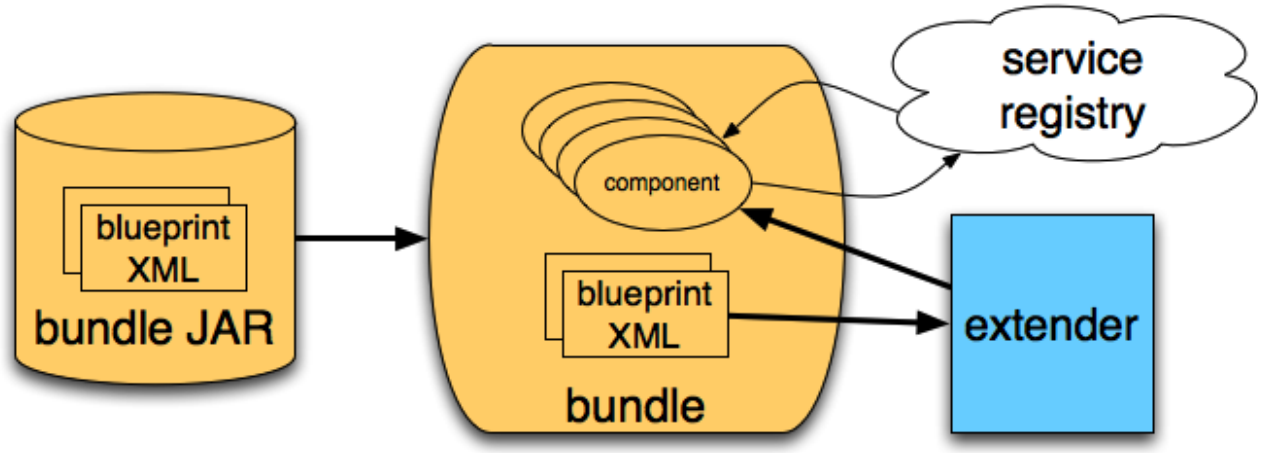

Figure 12: Gemini Blueprint

Gemini Management

Gemini Management is used to provide a JMX view of bundles in Virgo. Separate instances of Gemini Management are installed in the kernel and user regions in order to provide JMX access to both regions.

Logback

Logback is embedded in Virgo to provide its logging support. Standard logback configuration is supported. Virgo defines relatively high-volume trace logging for diagnostic purposes and relatively low-volume, internationalisable event logging for feedback to administrators.

Equinox OSGi Framework

Eclipse Equinox is embedded in Virgo as an OSGi framework implementation. Equinox is the reference implementation of the OSGi framework specification. Virgo uses several implementation specific features of Equinox to provide better OSGi bundle resolution diagnostics and to configure class loading and resource lookup to suit Virgo's requirements.

Equinox Services

Virgo embeds Equinox implementations of the OSGi standard Declarative Services for service publication and lookup, Configuration Admin for bundle configuration, and Event Admin for event propagation.

Equinox p2

Virgo uses p2 for initial provisioning of all its deliverables as an alternative to the regular ZIP file install which is also provided. Virgo nano and nano “full” support runtime provisioning by p2. Runtime provisioning is not yet supported for Virgo kernel-based distributions as p2 has no support for multiple regions.

Since Eclipse Ganymede (3.4) p2 has been the provisioning platform of choice for Eclipse and Eclipse-based applications. p2 introduced many features such as its own repository concept, artefact mirroring, general purpose provisioning, reversion functionality, and much more. Even though p2 has been used primarily for Eclipse provisioning and extending, the system had the potential to fit

in many environments and scenarios.
P2 supports three types of artefact: regular OSGi bundles, *features* which group bundles or other features together, and products which are self-contained units that groups bundles and features along with optional configuration for them. The configuration can include auto-start, start level, environment and system property settings.

p2 defines two types of repositories: metadata and artefact. Metadata repositories contain meta-information about the artefacts included in the repository: their requirements, provided capabilities, instructions applied at installation, etc. Artefact repositories contain the coordinates of the artefact binaries included in the repository. Normally these two repositories go together.

p2 provides a GUI or can run in headless mode. The GUI can be used to publish artefacts into a p2 repository or to install them as standalone entities or into an existing product. The p2 operations are also available as headless ant tasks which can be embedded in scripts.

Full-blown p2 provisioning is available only in Virgo Nano. The other Virgo distributions (Kernel and higher) can have their kernel region modified via p2, but not their user region.

Equinox Region Bundle

Virgo implemented a notion of region which is an isolated partition of the OSGi framework and implemented this using OSGi standard framework hooks provided by the Equinox framework. The region concept was of interest to others outside Virgo and so the implementation was moved into Equinox as the Equinox Region bundle which Virgo now embeds.

The Equinox Region bundle is also used by the Apache Aries implementation of the OSGi Subsystems specification as well as by others who wanted to implement multi-tenancy support directly on top of an OSGi framework.

Since the Equinox Region bundle depends only on standard OSGi features such as the framework hooks, it will run on any OSGi framework implementation which provides those features.

Standards

Virgo supports the following standards, usually by embedding other components as described above.

- Servlet Specification
- Java EE Web Profile
- OSGi Core
- OSGi Web Applications
- OSGi Blueprint
- OSGi Declarative Services
- OSGi Log Service
- OSGi Event Admin Service
- OSGi Configuration Admin Service

Although Virgo does not yet implement the OSGi Subsystems specification, many of the facilities of Subsystems are already part of Virgo including PARs, scoped and unscoped plans, and provisioning of dependencies from a repository. One notable advantages of the OSGi Subsystems specification over Virgo was the ability for multi-bundle applications to share their artefacts. This support was therefore added to Virgo 3.5.0 to provide equivalent function in a Virgo style. Support for standard subsystem artefact types in Virgo is a future plan item.

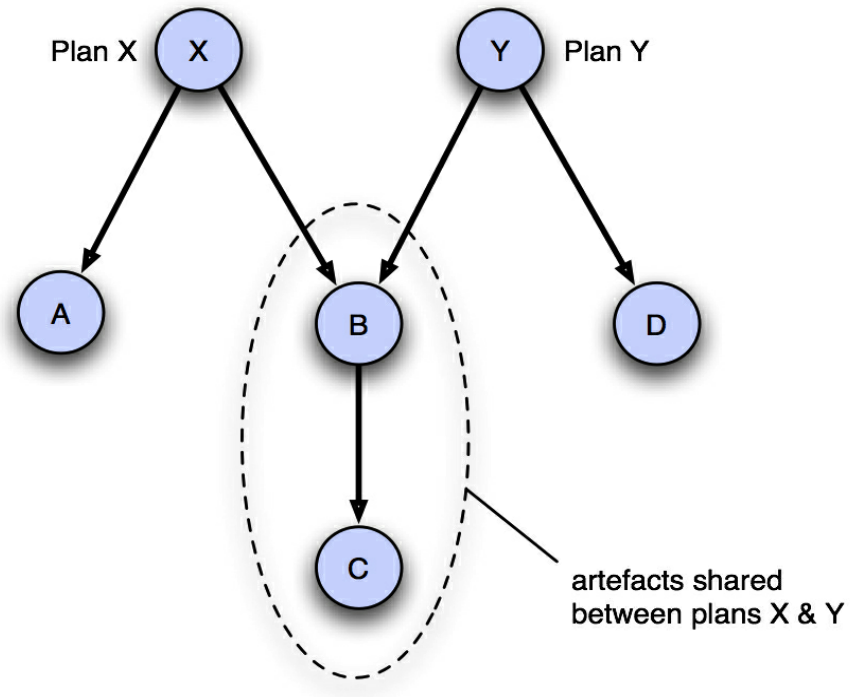

Figure 13: Sharing Artefacts Between Plans

Further Information

•Virgo home page at http://www.eclipse.org/virgo/

•Virgo documentation at http://www.eclipse.org/virgo/documentation/

◦ User Guide

◦ Programmer Guide

◦ Snaps Guide

•Greenpages sample application at http://www.eclipse.org/virgo/samples/

◦ Sample ready to build with Maven

◦ Greenpages Guide\

•Virgo Wiki at http://wiki.eclipse.org/Virgo

•Virgo FAQ at http://wiki.eclipse.org/Virgo/FAQ

•Virgo community forum at http://www.eclipse.org/forums/index.php?t=thread&frm_id=159

•Virgo bugs on bugzilla at https://bug.eclipse.org/bug.