Newton 1.4-b52: Developer Guide

Table of contents

1. Quick Start

1.1. Newton: Installation

1.1.1. Prerequsites

Newton requires a Java 5 or 6 Runtime Environment (JRE). The Java 7 pre-release versions are not supported.

> **Note:**
> If you do not have a Java 5 or 6 JRE on your PATH, you should set the environment variable JAVA_HOME to point to the desired JRE (or JDK).

Newton is a Java based application, hence it should run on any operating system with a compatible virtual machine installed. We have have currently tested Newton using Sun JVMs (except on Macs) on the following platforms:

- Linux
  - Red Hat Enterprise Linux 3, 4
  - Open SuSE 10.2
  - Fedora Core 4, 5
  - Open SuSE 10.2
- Windows
  - XP SP2
  - Server 2003 SP1
- Mac OS X
  - 10.4 Tiger
  - 10.5 Leopard

1.1.2. Installing

To install Newton you must download the distribution and extract the package.

Unpack the distribution in a directory of your choice:

```
cd $HOME
unzip newton_bin-1.3.x.zip
```

This creates the following structure under $HOME/newton-1.3.x:

bin
Script to start the Newton container and other utilities.
etc
Configuration files.
lib

OSGi bundle jar files.
example
Example bundles and source code.
doc
Documentation.
sdk
Software Development Kit.

1.1.3. Testing the container

You should now be able to start the Newton container:

```
$ cd $HOME
$ bin/container
+ exec java -javaagent:lib/core/components/newton-instrumentation.jar
-cp lib/launcher.jar:lib/equinox/ [truncated]
Loaded instrumentation sun.instrument.InstrumentationImpl@e0bc08
Launcher:DEBUG: Reading /Users/derek/.container.ini
Launcher:DEBUG: Reading
/Volumes/Users/derek/eclipse/Newton/build/install/etc/container.ini
Launcher:DEBUG: Reading
/Volumes/Users/derek/eclipse/Newton/build/install/etc/config.ini
Launcher:DEBUG: Reading
/Volumes/Users/derek/eclipse/Newton/build/install/etc/config-equinox.ini
Launcher:DEBUG: lockfile not present:
/Users/derek/Newton/build/install/var/lock
Launcher:INFO: org.eclipse.core.runtime.adaptor.EclipseStarter []
Application[newton] Install org.knopflerfish.bundle.log success
Application[newton] Install org.knopflerfish.bundle.cm success

[output truncated]

Executing: installer install etc/instances/presence-service.composite
Executing: installer wait etc/instances/presence-service.composite
active
Boot complete
>
```

Congratulations, you have successfully installed Newton.

To shutdown the container, type:

```
> shutdown
Application[newton] Stop org.cauldron.newton.boot success
Application[newton] Stop org.cauldron.newton.init success

[output truncated]

Application[newton] Uninstall org.cauldron.newton.install.session
success
Application[newton] Uninstall org.knopflerfish.bundle.consoletty success
Application[newton] Stopped
$
```

2. Concepts

2.1. Newton: Component Model

2.1.1. Motivation

Newton provides a distributed component model.

To be useful, a distributed component model needs to address the fundamental differences between local and distributed computing, as famously expressed in The Eight Fallacies of Distributed Computing (http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing) and A note on distributed computing (http://research.sun.com/techrep/1994/abstract-29.html) .

Another consideration is ease of development. The recent plethora of POJO based application frameworks such as Spring, and their rapid adoption by the development community serve to highlight the advantages of lightweight development models. That is, separation of domain model from infrastructure concerns, increased productivity and maintainability, and reduced vendor lock-in.

It is with these considerations in mind that Newton has been developed. The objective is to provide a lightweight development model for distributed components in which the unavoidable realities of distributed computing are directly addressed.

2.1.2. Key technologies

The world in which a distributed component lives is a highly dynamic one, subject to unforeseen failures and network state ambiguities. Also administration of distributed systems, for example deployment and clean-up of resources, is currently an onerous task, and stands in the way of all efforts to develop autonomous distributed systems. Two key technologies Newton uses to address these concerns are OSGi, which is central to the whole Newton component model, and Jini, which is the basis of Newton's remoting infrastructure. Newton also makes use of SCA to describe describe assemblies of componenents, or *composites* in SCA terminology.

2.1.2.1. OSGi

OSGi provides a highly dynamic and well designed services model for a single JVM. OSGi's unit of deployment, the Bundle, uses a state of the art peer classloading model, rather than a traditional hierarchical classloading model. This enables OSGi to guarantee a consistent class view across the container and avoids the classloader related class mismatch errors that plague hierarchical approaches. Bundles also have a well defined lifecycle from install to uninstall. OSGI bundles also improve on the encapsulation provided by traditional jar files, publically exposing only those classes which form part of their intended API, and keeping the rest private. Physically bundles are just jar files with

extra metadata, via which inter bundle dependencies can be resolved.

OSGi also makes use of an internal services registry, via which services created during the lifetime of different bundles can find each other.

Additionally, OSGi has a well thought through security model at both Bundle and Service level.

OSGi's rich and highly dynamic model made it a natural choice for Newton, in which the container itself is built out of OSGi bundles, and in which end user component code is deployed in bundles.

At present Newton runs in an OSGi R4 container. Knopflerfish and Equinox are supported directly. We expect to add Felix to this list before too long.

2.1.2.2. Jini

Jini provides a service oriented model that explicitly addresses the realities of distributed computing. In particular Jini provides a distributed registry, via which remote services can find each other, and makes extensive use of leasing to ensure reclamation of resources that are not explicitly release by failed clients. Jini also provides a high quality extensible RMI stack, and strong support for network level and remote code security.

Newton uses Jini's remote registry to track and wire up the remote dependencies of Newton components. Future releases of Newton will make use of Jini's distributed transaction management capabilities and of Javaspaces for distributed process coordination.

2.1.2.3. SCA

SCA is a language, runtime and protocol agnostic component wireup standard. Essentially it allows for the specification of a hierarchical assembly of components, either within a single runtime, or distributed across disparate systems.

Newton's implementation of SCA is highly dynamic. Starting with an SCA descriptor, Newton is able to instantiate a distributed system across available hardware, provisioning all necessary resources and wiring up the SCA composites without any need for administrator assistance. During its lifetime Newton monitors the system and responds to problems. Newton also monitors the SCA descriptor, propagating changes to this document onto the deployed system. When the SCA system is no longer required Newton is able to completely deprovision it and to release all of the resources it used.

2.1.3. Component model

2.1.3.1. Overview

The minimal unit of deployment in Newton is an SCA composite running within a single

process with code backed by a single OSGi bundle known as a factory bundle. Factory bundles either contain or import all the class and resource files necessary to instantiate composites of a given type, and can be used to create many composite instances of this type. The way in which a factory bundle creates and manages its composites is specified by the *composite template*(s) it contains. Composite templates are SCA descriptors specifying the overall structure of the composites a given factory bundle can create, including their lifecycle management, provided services and required references.

To install a Newton composite its instance descriptor is given to the Newton installer. This descriptor is also SCA based, and modifies a composite template, overriding it with instance specific information. The composite is removed when its descriptor is removed.

Newton composites can provide and refer to multiple services. Throughout a composite's lifetime Newton monitors these and wires them together based on their binding type and the metadata with which they are annotated. Newton's bindings notice when a service dependency is lost due to failure, uninstallation or disconnection, and respond by trying to rewire the dependency, perhaps to an alternative service.

When the first Newton composite of a given type is installed Newton installs the corresponding factory bundle and all of its dependencies. When the last composite that type is uninstalled Newton's bundle garbage collection mechanism removes any bundles that are no longer in use.

Higher level SCA composites and systems can be built out of the minimal composites backed by factory bundles. Newton manages these larger composites as follows:

Given an SCA based description of some *target state*, Newton deploys, instantiates and wires up all necessary composites to reach this state. Newton then monitors the deployed system for deviations from this target state and works to get back to it. For example, if Newton notices a composite has failed it will respond by reprovisioning a new one. Newton's notion of target state is richer than simple composite counting, and can be based on any available system metrics, so that for example, part of the target state could be an invocation rate SLA.

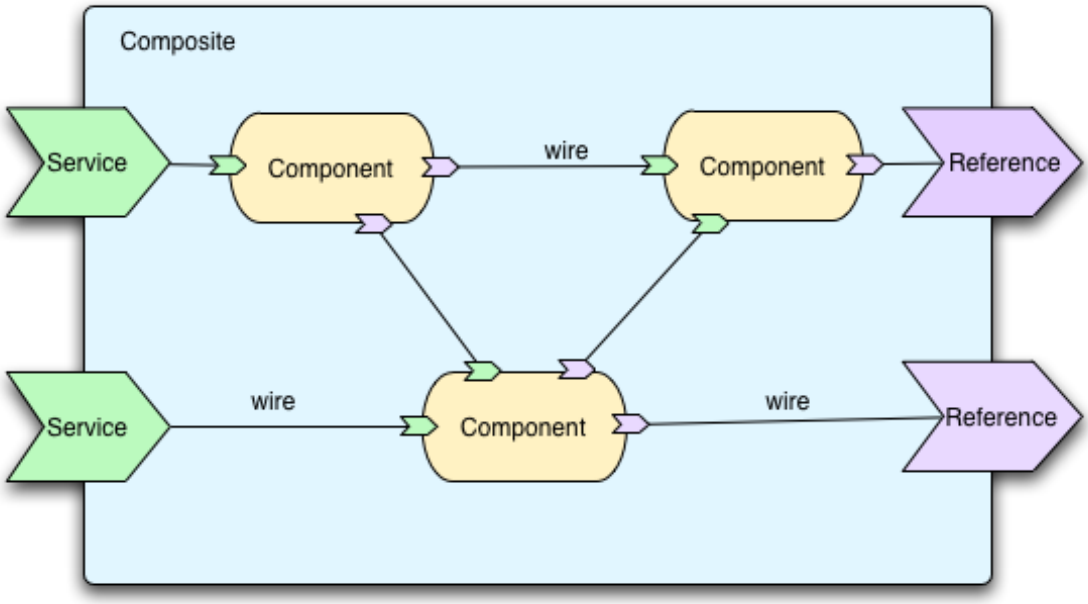
Bundles and and other files are made available to Newton when they are added to the Content Distribution System (CDS). Each item of CDS *content* belongs to a *zone*. Content in the *boot* zone is private to each container instance and includes, for example, those bundles required during the bootstrap phase, before the remote part of Newton is active. On the other hand all Newton containers belonging to a given *Fabric* (i.e. multi-container Newton system) have the same view of Content in CDS's *remote* zone. Since the remote zone is visible across multiple containers, any bundle stored in it can be installed on demand by any container in the Fabric.

2.1.3.2. Composite Structure

Newton's minimal deployable units are non nested SCA composites containing

components which wrap the composite's service implementations. Components have an implementation type which describes how to wrap a particular kind of underlying implementation. At the moment Newton provides implementations based on POJOs and on Jini Starter Service services. However, the mechanism is highly extensible, as illustrated by the Spring implementation type which was initially developed as an optional add-in for Newton, but is now bundled as standard.

Composites provide *services* to other composites and have *references* which express their own external service requirements. The technology used to connect services and references is expressed by their *binding*. At the moment Newton supports local OSGi and remote Jini based bindings, although users are shielded from most of their complexity. Components within the same composite can also refer to each others' services directly. The following diagram illustrates this model.



On this diagram the internal links represent wires between services and references within a single component.

Within Newton, service and reference bindings are annotated with binding specific metadata. The OSGi and Jini based bindings that Newton currently supports have a shared approach to metadata. Each annotates its services with multiple name-value pair attributes, in which the values are wrappers around simple Java types. On the other hand, references are annotated with LDAP style filters over service attributes. Newton connects references to services with attributes satisfying their LDAP filter.

The number of matching services a reference wants to be connected to is controlled by the import's *multiplicity*. Multiplicity examples are 1..1 for a mandatory service, 0..1 for an optional one, and 0..* when all matches are sought.

The structure of a given composite type, and the way in which it creates its components is specified by its SCA based *composite template descriptor*. Individual composites are described by SCA based *composite instance descriptors* referencing and overriding this template. A descriptor example is shown below - this one is taken from the Chainlink Demo.

```
<?xml version="1.0"?>
<composite name="braceLinkTemplate">
    <description>chainlink demo brace link</description>

    <reference name="nextLink" multiplicity="0..1">
        <interface.java
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link"/>
        <binding.osgi/>
    </reference>

    <service name="brace-link-export">
        <interface.java
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link"/>
        <binding.osgi/>
    </service>

    <component name="brace-link">
        <description>A brace decorated chain link</description>
        <property name="linkName" value="Unnamed" type="string"/>
        <reference name="nextLink"/>
        <implementation.java.callback
impl="org.cauldron.newton.test.bundles.chainlink.brace.BraceLink"/>
    </component>

    <wire>
        <source.uri>nextLink</source.uri>
        <target.uri>brace-link/nextLink</target.uri>
    </wire>

    <wire>
        <source.uri>brace-link</source.uri>
        <target.uri>brace-link-export</target.uri>
    </wire>
</composite>
```

This composite has a reference to an interface called `nextLink` of type `org.cauldron.newton.test.bundles.chainlink.interfaces.Link` using the OSGi binding. If a suitable service can be found then the reference will be wired to it, but the multiplicity constraint of `0..1` means that it can function without one. If there was an LDAP filter on the reference it would be on the OSGi binding; we'll see one in a moment in the instance descriptor.

This composite creates one component called "brace-link". This component is wired to the brace-link-export service. It also requires a service called bracelink/nextLink, which is wired to the nextLink reference. The `<property ..>` tag specifies a configuration value which can be used by the service implementation.

This component creates and manages its dependencies using a POJO style "callback" lifecycle. The callback lifecycle creates components by instantiating the `impl` class using

either an empty constructor or one taking a map holding the config values specified in the component's `<config ..>` tags. The callback lifecycle manages service dependencies by calling `add<requirement-name>` and `remove<requirement-name>` methods on the impl class. The `impl` class corresponding to this template is shown below.

```
package org.cauldron.newton.test.bundles.chainlink.brace;

import java.util.Map;
import org.apache.log4j.Logger;
import org.cauldron.newton.test.bundles.chainlink.interfaces.Link;

/**
 * Callback style POJO component impl
 */
public class BraceLink implements Link {
    private static final Logger log = Logger.getLogger(BraceLink.class);

    private String linkName;

    private Link next;

    //the service config is passed to the constructor and used to
    //determine the link name
    public BraceLink(Map config) {
        linkName = (String) config.get("linkName");
    }

    //invoked by the callback lifecycle whenever a link is connected
    public void addNextLink(Link link) {
        next = link;
    }

    //invoked by the callback lifecycle whenever a link is lost
    public void removeNextLink(Link link) {
        next = null;
    }

    public String append(String message) {
        if (log.isDebugEnabled())
            log.debug("appending [" + message + "] in " + linkName);

        if (next != null && message.length() < 4096) {
            return next.append(message + "->{" + linkName + "}");
        }
        else {
            return message + "->{" + linkName + "}" + " [end of chain]";
        }
    }

    //invoked by the callback lifecycle if available
    public void destroy() {
        log.debug("DESTROYED BraceLink: " + linkName);
    }
}
```

To actually create an instance of a composite an SCA based instance descriptor is passed to the Newton installer. An instance descriptor corresponding to the above template is

shown below.

```
<?xml version="1.0"?>
<composite name="LinkA">

    <bundle.root bundle="chainlink-local-brace-bundle" version="1.0"/>
    <include name="braceLinkTemplate"/>

    <reference name="nextLink">
        <binding.osgi filter="(tag=b)"/>
    </reference>

    <component name="brace-link">
        <description>link A</description>
        <property name="linkName" value="A-link" type="string"/>
    </component>

    <service name="brace-link-export">
        <binding.osgi>
            <attribute name="tag" value="a" type="string"/>
        </binding.osgi>
    </service>

</composite>
```

Here the `<bundle.root ..>` element specifies the factory bundle containing (or importing) the code needed to create the composite. The next line includes the template that this descriptor will override with instance information. The rest of the descriptor specifies instance specific binding information and configuration properties. Note how the reference binding now has an associated LDAP filter of `(tag=b)`, the service binding is now annotated with `tag=a`, and the `linkName` property value has been redefined to give the instance a name.

2.1.3.3. Composite Lifecycle

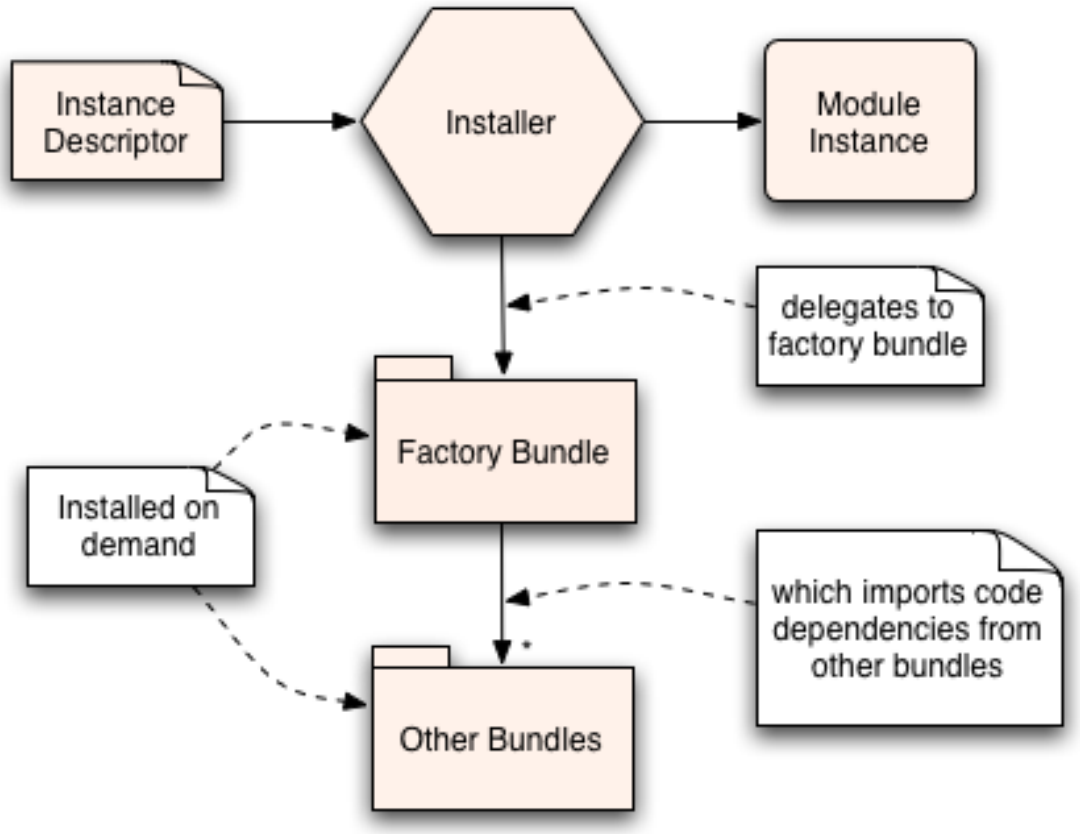
When Newton composites are being installed, all that needs to be specified is their composite instance descriptor. Neither end users, nor higher level Newton facilities, such as those managing full SCA systems, need to be concerned about installing or uninstalling the bundles containing the composites's class and resource files. During composite installation the installer identifies the composite's factory bundle from its descriptor and delegates to it for the rest of the install process. It is important to understand that when the request to install a composite is received, the corresponding factory bundle may not be present. Newton solves this problem as follows.

Newton tracks bundle dependencies by maintaining an OBR index of all bundles in CDS. When the first instance decriptor of given type is passed to the installer, it uses this index to check that the factory bundle and all of its dependencies are installed, and then installs any that are missing. Next the factory bundle is started, after which it serves as a factory for turning instance descriptors corresponding to its template(s) into runtime composites.

For completeness we note that Newton uses a bundle garbage collection (BGC) mechanism to remove unused bundles. In the current case factory bundles are marked as

BGC roots, which prevents both their removal and the removal of their dependencies by the BGC. After the last instance descriptor of a given type is removed from the installer, that composite type's factory bundle is stopped and its BGC root status is removed. This makes it and its dependencies eligible for removal by the BGC if they are not dependencies of another BGC root.

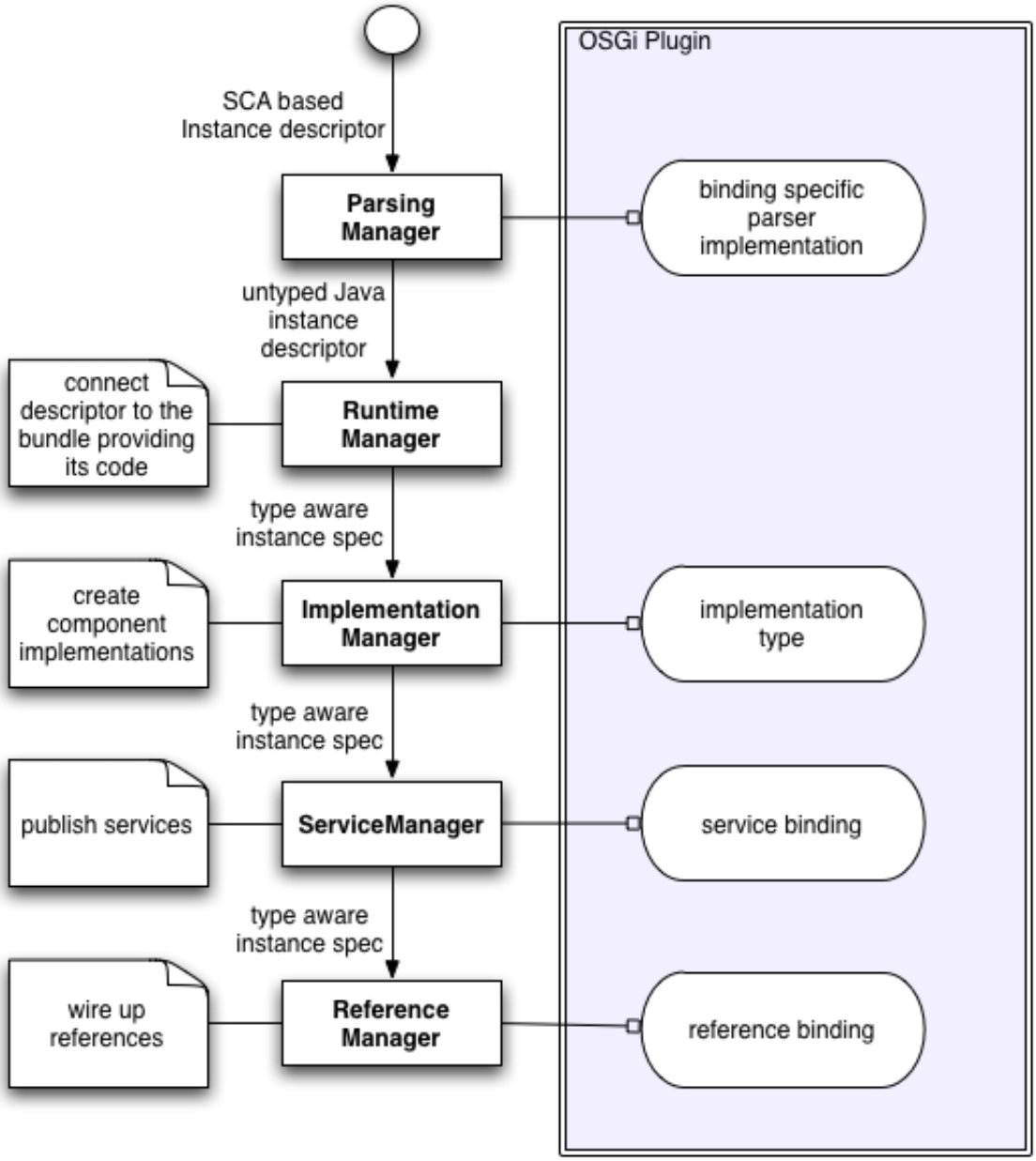
The install process just described is illustrated below.



2.1.3.4. Extensible composite instantiation pipeline.

The processes described above are all part of Newton's composite instantiation pipeline. This pipeline is extensible at several points, so that it can be adapted to support different component implementation types and transport bindings. These extensions can be implemented using Newton composites or non-Newton OSGi services.

The following diagram provides a simplified view of this pipeline and its extensibility points.



The steps shown correspond to the following

1. The parser converts an XML based instance descriptor into a Java runtime representation of the same information. Parsing is extensible so that it can read any configuration information that used by the plugins to the other extensibility points.
2. The runtime manager connects the SCA instance descriptor to the code needed to create the composite (using the process described earlier in this document) and uses it constrain the corresponding SCA template.

3. The resulting SCA descriptor is turned into into a runtime instance of the composite. This involves creating all of the components in the composite using implementation type specific approaches. At the moment there are three implementation types. One for POJO style components with callback based lifecycles, another for POJO style components that are passed their dependencies at construction time, and a third that wraps Jini Starter Service based services. A Spring implementation type which demonstates the pluggable nature of implementation types is also available.
4. The service bindings publish the component's services. These bindings are also easily exstensible. At the moment there are bindings for the OSGi and Jini. Future Newton releases are likely to support additional bindings, some based on alternate protocols, others on quality of service concerns.
5. The references bindings track and wire up the component's references, rewiring them in response to change. These bindings are also easily exstensible. At the moment there are bindings for OSGi and Jini. Future Newton releases are likely to support additional bindings, some based on alternate protocols, others on quality of service concerns.

2.2. Provisioning

2.2.1. Newton System Manager

2.2.1.1. Overview

The Newton System Manager is used to manage how many and which sort of SCA composites are installed in a Newton Fabric. The system manager is built using a collection of sub-components built from SCA documents.

A specific system manager can manage many systems. The system manager uses a document called a System Descriptor to define the set of composites that make up a system.

The system manager has two lifecycle events related to systems; manage and retire. The first time a system manager is asked to manage a system it reads the system document and builds the set of composites that it defines. It then passes these composites to the provisioner for installation somewhere within the Fabric.

If a system administrator wants to make a change to a given system he/she simply updates the system document and asks the system manager to manage the new document. A system manager treats two system documents as referring to the same system if their names match.

On update the system manager figures out the delta between the current deployment and the new requirement and actions the change only. So if a system is being increased from one to ten nodes then nine new instances are created and the existing one is reused. If it is being reduced in size from ten nodes to five nodes then five nodes are picked for uninstall and the rest remain in place.

When the system administrator no longer wants the system manager to deploy a specific system he/she simply retires the system and the system manager garbage collects all resources previously allocated.

Importantly the system document does not have to specify a static set of composites. It is possible to build dynamic behaviours by extending the ReplicationHandler interface and plugging the replication handler into the local osgi registry of the Newton process.

ReplicationHandlers are able to dynamically change the number of composites based on external information (such as management level information). The system manager will treat changes from replication handlers in the same way as changes from the administrator and simply deploy or undeploy the delta.

2.2.1.2. Example system descriptor

The following xml document is used to define a static set of six composite instances: one

CompositeA; two CompositeB; and three CompositeC instances.

```
<system name="example">
  <description>An example static system</description>
  <system.composite name="a" bundle="example-bundle"
template="CompositeA" version="1.0" />
  <system.composite name="b" bundle="example-bundle"
template="CompositeB" version="1.0">
    <replicator name="two" />
  </composite>
  <system.composite name="c" bundle="example-bundle"
template="CompositeC" version="1.0">
    <replicator name="three" />
  </composite>
  <replication.handler name="two"
type="org.cauldron.newton.system.replication.FixedSizeReplicationHandler">
   <property name="size" value="2" type="integer"/>
  </replication.handler>
  <replication.handler name="three"
type="org.cauldron.newton.system.replication.FixedSizeReplicationHandler">
   <property name="size" value="3" type="integer"/>
  </replication.handler>
</system>
```

Each composite element specifies four attributes:

- The name attribute defines a unique name for this composite within the system.
- The bundle attribute specifies the composite bundle that provides this composite - must be equal to the value specified in that bundles Bundle-Symbolic-Name jar manifest attribute.
- The template attribute specifies the template within that composite bundle to use - must be equal to the name attribute in the composite template xml file.
- The version attribute specifies the version of the composite bundle to use.

The first composite is an empty xml element, this is a short hand way of referring to a fixed size replicator with a size config of 1.

The second two composites include a replicator pointer which point to named replication elements in the document below. A replicator can be reused by more than one composite so if for example you wanted to have three CompositeB instances you could change the replicator element within the b composite to look like:

```
<system.composite name="b" bundle="example-bundle" template="CompositeB"
version="1.0">
  <replicator name="three" />
</composite>
```

What if you want to be able to scale the number of composites with the number of available containers though? In this case you can't use a fixed size replicator else you'll be constantly modifying you system declaration. Instead you should use a scalable replication handler:

```
<replication.handler name="scale"
type="org.cauldron.newton.system.replication.ScalableReplicationHandler">
```

```
 <property name="scaleFactor" value="1" type="float"/>
</replication.handler>
```

This will create a replicator that will match the number of composite instances to the number of available containers. If you wanted to take up half the available containers specify a scaleFactor of 0.5, if you wanted two instances per container specify 2.0, etc.

2.2.2. Provisioning Service

2.2.2.1. Overview

The provisioner service is used to install an SCA composite in a Newton instance somewhere within a Newton fabric.

When an SCA composite is submitted to the provisioner, it first sends a hosting requirement containing the composite definition to each existing Newton instance within the fabric. Each Newton instance then assesses the composite and responds with a “cost” to host the composite. The provisioner then installs the composite in the Newton instance that offers the lowest cost and for the highest value.

The provisioner is designed to work in highly dynamic infrastructures where instances may come and go. If a Newton instance that was previously hosting a composite either ejects the composite, fails or is shutdown then the provisioner will simply re-negotiate with the remaining instances to find a suitable fall back position.

Periodically if there is nothing else for the provisioner to do it will attempt to re-negotiate composites that are already installed. If a “better” host is found for a composite then the provisioner will instantiate a new instance of the composite on the new container and garbage collect the old instance.

2.2.2.2. Future Work

Currently the notion of which instance is “better” than another one is handled via a pluggable Java API called an Evaluator. However this is a global view of “better” which may not be valid across all types of composite. Much better to allow each composite to define it's own notion of better (falling back to some default view if not specified?) and negotiate on this basis.

This will require a mechanism to mark up in the SCA composite the view of “better” and an implementation of an evaluator that knows how to read this marked up info and evaluate returned containers accordingly.

2.2.3. Replication Handlers

2.2.3.1. Overview

Newton provides an extensible mechanism for specifying how many of a given composite

should be deployed to a Newton fabric. This mechanism is via the ReplicationHandler and ReplicationHandlerFactory interfaces. ReplicationHandlers can provide dynamic scalability to composites within a Newton fabric.

2.2.3.2. API

In order to build a replication handler you should implement the following interfaces:

```
package org.cauldron.newton.system.api;
import java.util.Collection;
import org.cauldron.newton.model.component.CompositeInstance;
import org.cauldron.newton.model.system.SystemCompositeDescriptor;
public interface ReplicationHandler {
    void initialise(SystemContext ctx) throws
SystemConfigurationException;
    SystemContext getSystemContext() throws
SystemConfigurationException;
    Collection<CompositeInstance> replicate(SystemCompositeDescriptor
template, SystemStateModel model) throws SystemConfigurationException;
    void destroy(SystemContext ctx);
}
```

```
package org.cauldron.newton.system.api;
import org.cauldron.newton.model.system.ReplicationDescriptor;
public interface ReplicationHandlerFactory {
    ReplicationHandler createNewHandler(ReplicationDescriptor
descriptor) throws SystemConfigurationException;
}
```

The ReplicationHandler does the actual work of building composites based on templates supplied from the system document. The ReplicationHandlerFactory is registered in the OSGi registry of the Newton instance running the system manager – this creates instances of ReplicationHandlers based on the configuration supplied via the ReplicationDescriptor element.

2.2.3.3. Example

The following example shows a ReplicationHandler that deploys composites only at specified times of the day:

```
package com.example.replication;
import java.util.Calendar;
import java.util.Date;
import java.util.Map;
import java.util.Timer;
import java.util.TimerTask;
import org.cauldron.newton.system.replication.SimpleReplicationHandler;
import org.cauldron.newton.system.api.ReplicationHandler;
import org.cauldron.newton.system.api.SystemConfigurationException;
import org.cauldron.newton.system.api.SystemContext;
public class WorkingDayReplicationHandler extends
SimpleReplicationHandler implements ReplicationHandler {
        private int count;
        private int hourStart;
        private int hourEnd;
```

```
        private Timer timer;

        public WorkingDayReplicationHandler(Map<?, ?> config) {
                Integer i = (Integer) config.get("count");

                if ( i != null ) {
                        count = i;
                }

                i = (Integer) config.get("hourStart");

                if ( i != null ) {
                        hourStart = i;
                }

                i = (Integer) config.get("hourEnd");

                if ( i != null ) {
                        hourEnd = i;
                }
        }
        public void initialise(SystemContext ctx) throws
SystemConfigurationException {
                super.initialise(ctx);
                timer = new Timer();
                scheduleRecheck();
        }
        public void destroy(SystemContext ctx) {
                timer.cancel();
                timer = null;
                super.destroy( ctx );
        }
        @Override
        protected int calculateRequired() throws
SystemConfigurationException {
                return isWorkingHours() ? count : 0;
        }

        private boolean isWorkingHours() {
                int hour =
Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
                return hourStart < hour && hourEnd > hour;
        }
        private void scheduleRecheck() {
                TimerTask recheck = new TimerTask() {
                        @Override
                        public void run() {
                                try {
                                        getSystemContext().rebuild(0);
                                        scheduleRecheck();
                                } catch (SystemConfigurationException e)
{
                                        e.printStackTrace();
                                }
                        }
                };

                if ( isWorkingHours() ) {
                        timer.schedule(recheck, nextCheck(hourEnd));
```

```
                }
                else {
                        timer.schedule(recheck, nextCheck(hourStart));
                }
        }
        private Date nextCheck(int hour) {
                Calendar c = Calendar.getInstance();

                if (c.get(Calendar.HOUR_OF_DAY) > hour) {
                        c.add(Calendar.DAY_OF_YEAR, 1);
                }

                c.set(Calendar.HOUR_OF_DAY, hour);

                return c.getTime();
        }
}
```

This replication handler is then plugged into the system manager via the following factory:

```
package org.cauldron.newton.system.replication;
import java.util.Map;
import org.cauldron.newton.model.system.ReplicationDescriptor;
import org.cauldron.newton.system.api.ReplicationHandler;
import org.cauldron.newton.system.api.ReplicationHandlerFactory;
import org.cauldron.newton.system.api.SystemConfigurationException;
public class ExampleReplicationHandlerFactory implements
ReplicationHandlerFactory {
        public ReplicationHandler createNewHandler(ReplicationDescriptor
descriptor) {
                if ( "com.example.workday".equals( descriptor.getType()
) ) {
                        Map<?,?> config = descriptor.getConfig();
                        return new WorkingDayReplicationHandler( config
);
                }
                else {
                        return null;
                }
        }
}
```

To use this replication handler you would use the following mark up in a system document:

```
<replication.handler name="workday" type="com.example.workday">
 <property name="count" value="10" type="integer"/>
 <property name="hourStart" value="9" type="integer"/>
 <property name="hourEnd" value="17" type="integer"/>
</replication.handler>
```

2.2.4. Contracts And Features

Contracts, features and assessors are the mechanism via which a Newton Provisioner negotiates with other Newton instances to find a location to install a composite: features

and assessors are properties of a specific Newton instance; contracts are associated with a specific composite.

Another way of saying this is that contracts specify the things a developer knows the composite requires in order to run, where as features and assessors specify the system administrators ability to provide such features on a given Newton instance.

2.2.4.1. Features

Newton instances are attributed with a list of features, these are key value pairs that describe details about the overall environment the instance provides. The list of features is extensible via the FeaturesProvider interface which is aggregated into the instance via the local OSGi registry.

2.2.4.2. Assessment

The CompositeAssesor interface is used to check whether or not a specified Composite may be installed within a Newton instance. CompositeAssessors are aggregated into the instance from the local OSGi registry.

Assessors are chained together in a list ordered via priority (highest priority first). Each assessor is queried in turn as to the cost - expressed as an integer value where -1 <= cost <= Integer.MAX_VALUE – to host a composite based on the list of current installs.

If -1 is returned this effectively means the cost is infinite and the composite should not be installed here at all and causes the Newton instance to stop querying the chain of assessors and simply rejects the composite install.

Any other value is summed onto the aggregate cost from all assessors in the chain. The total cost is returned to the provisioner which then installs the composite to the instance that provides the lowest cost.

2.2.4.3. Contract

Contracts allow the developer or user of a composite to specify details about how and where a composite should be installed. Contracts are marked up in system documents as xml elements of the following form:

```
<contract features="" cancelationCost="" lifetime="" />
```

- Features – an ldap expression evaluated against the list of features for the Newton instance, must evaluate to true for the composite to be installed on this instance. e.g. `(&(machine.arch=i386)(machine.availableProcessors>1))` specifies that the composite may only be installed on a Newton instance running on a multi CPU, Intel based architecture.
- cancelationCost – an integer value that specifies whether or not a composite should be rebalanced (i.e. moved to another Newton instance) if a lower cost installation

becomes available during the lifetime of the composite install. The value -1 may be used to specify that the composite should never be rebalanced. All other values are then used to calculate the required drop in cost before a composite will migrate to a new instance. The default is 1.

- lifetime – this is a crude measure of how long a composite install should last – it is a fixed time in milliseconds after which the composite will be uninstalled from the container. This is an area of active development to extend the notion of lifetime for the next release so expect much richer patterns in the near future.

2.3. Installation

2.3.1. Installation Lifecycle

In Newton there are a number of lifecycle phases involved in the process of installing and uninstalling a composite, namely; resolution, download, check, activate, deactivate, garbage collect.

2.3.1.1. Resolution

During this phase all requirements specified by the composite are matched to resources within the fabric. Specifically this means:

- walking through the list of SCA includes specified in the composite and applying them to the current composite (this may be recursive)
- walking through the list of OSGi bundles that are specified via bundle imports and exports with a preference to bundles already installed in the current Newton instance
- adding any implied OSGi bundle dependencies (e.g. spring libraries for <implementation.spring>)

Note any one of these phases may fail due to lack of resources at runtime, however if this stage completes successfully installation passes onto the download phase.

This phase significantly eases the work of system administrators when trying to figure out the list of bundles that are needed to support a given application as Newton handles this automatically without operator intervention.

2.3.1.2. Download

Having resolved the list of bundles that need installation any that are not currently downloaded are pulled from CDS.

2.3.1.3. Validate

Once all resources have been downloaded Newton uses a pluggable set of RuntimeActivationGuards to decide whether or not a composite should be activated.

Guards can delay activation till external criteria are met.

2.3.1.4. Activate

Once all resources have been resolved, downloaded and the composite has been validated, Newton starts to construct the composite. This happens in the following order:

1. Implementation objects are instantiated via their implicit construction mechanism
2. Reference bindings are built and wired into the implementation
3. Service bindings are built, wired to their respective implementations.
4. Lifecycle callbacks are made to the component implementations.

2.3.1.5. Running

During this phase Newton monitors the references required by the composite and informs it of changes to services supplying those references. If all references requirements are met then Newton publishes the Service interface such that it can be consumed by other references.

Periodically Newton will recheck the set of RuntimeActivationGuards installed in the Newton process and may deactivate the composite if a guard specifies that the composite should no longer be running.

2.3.1.6. Deactivate

The composite is removed from the Newton runtime, services are shutdown, references released and implementations destroyed. In this state the OSGi bundles downloaded to support a given composite install are still available on the local machine. OSGi bundles are only removed when all referencing composites have been uninstalled. This process is managed via a garbage collection scheme.

2.3.1.7. Garbage Collect

It is possible for more than one composite to make use of code from a given bundle, therefore though it is valid to install a bundle when a composite is installed it may not be valid to uninstall it when the same composite is uninstalled, due to it being in use by another composite.

Newton uses a reachability graph scheme similar to that found in the Java virtual machine to decide when a bundle may be safely finally removed frpm the runtime environment. This happens seamlessly and without the need for administrator intervention.

2.3.2. Presence Patterns

Presence is an implementation of the RuntimeActivationGuard pattern in Newton. It is used to implement behaviours such as fast failover strategies though mechanisms such as

active passive and buddy.

2.3.2.1. Active Passive

This strategy is used to ensure that a specified number of composite are active and another number is passive. A passive composite is activated if an active composite is lost (for example due to failure or administrative intervention).

The implementation used in Newton uses the concept of absolute time (as measured from the system clock of the machine on which the Newton process is running) of install to decide which composites are active. The composites that were installed first are activated and others remain passive.

Absolute time is used because it ensures stability in the fabric, other schemes such as a random or merit based schemes can also work but are prone to phases of instability – a new newton process may start up which effectively “usurps” control from existing composites. Note in some scenarios this may be desirable, likely hood is we will support some of these patterns in the relatively near future.

Example

Within an SCA composite specify the following xml elements:

```
<presence-group labels="myPresenceGroup"
strategy="org.cauldron.newton.presence.strategy.ActivePassiveStrategy">
  <property name="maximumActive" value="1" type="integer" />
</presence-group>
```

This specifies that a maximum of one composite should be active at any one time, all other installs will be in a passive state until the primary is uninstalled or lost.

2.3.2.2. Buddy

The buddy strategy links the activation of one composite to that of another, if one composite is activated so is the other, if the parent composite is deactivated so is the other. This allows for simple chaining of activation/deactivation life-cycles between components.

Example

Within an SCA composite specify the following xml elements:

```
<presence-group
strategy="org.cauldron.newton.presence.strategy.BuddyStrategy">
  <property name="buddy" value="myPresenceGroup" type="string" />
</presence-group>
```

This specifies that the composite is buddied to any composites which declare the label myPresenceGroup, if they activate, then this composite will activate.

Note:

By default this only applies to composites in the same Newton process. If you wish to buddy your composite to one potentially installed in another Newton process add the configuration property localOnly=false (type boolean)

2.4. Bindings

2.4.1. Custom Bindings

The Newton framework is designed to be an extensible system that allows implementors to provide implementations of bindings to a range of other service orientated protocols.

2.4.1.1. Binding Architecture

The starting point for a developer wishing to build a new binding for Newton are the interfaces:

```
org.cauldron.newton.component.factory.api.ReferenceBindingFactory
org.cauldron.newton.component.factory.api.ServiceBindingFactory
```

These two factories provide the entry point used by the Newton framework to bind POJO's within the JVM to external services. The ReferenceBindingFactory is called on by the Newton framework to import existing external services into the Newton world. Conversely the ServiceBindingFactory is used to export components from Newton as external services to the outside world.

When an SCA composite is installed in the Newton container the binding layer introspects each service and reference binding in the composite document - it looks for a corresponding factory that knows how to create that binding. Using the corresponding factory an instance of a ReferenceBinding or ServiceBinding is created that handles communicating the state of the service to the Newton framework.

A ReferenceBindingFactory's primary job is to create a ReferenceBinding object which provides two functions. Firstly it communicates the state of an external service to the Newton infrastructure and secondly it provides a mechanism to grab a proxy object (or the object itself if the SOA is an internal jvm model) to the service the reference is bound.

A ServiceBindingFactory's primary job is to create a ServiceBinding which listens for lifecycle events from the Newton infrastructure and publishes an external interface to the component that the service is bound to when the Newton infrastructure prompts.

2.4.1.2. Installing A Binding Factory

There are two parts to installing a binding factory.

1. Install the XML parser that knows how to interpret the XML elements
2. Install a reference or service binding factory that interprets the parsed elements and constructs the actual binding.

There are two parts to this process as we wished to disconnect the model for parsing

XML elements into a programmatic model of the composite from the process of parsing that model into a real implementation. This allows us to do interesting things like programatically manipulate a model after it has been read in from XML but prior to constructing a real implementation.

XML To Model

The first stage then is to publish a service to the osgi registry that implements either:

```
org.cauldron.Newton.descriptor.xml.api.ElementParser
org.cauldron.Newton.descriptor.xml.api.ElementParserGroup
```

This can be achieved in a composite using the following pattern:

```
<composite name="binding-parser">
        <service name="parser-export">
                <interface.java
interface="org.cauldron.Newton.descriptor.xml.api.ElementParser" />
                <binding.osgi />
        </service>
        <component name="parser-impl">
                <implementation.java.callback
impl="com.example.parser.MyBindingParser" />
        </component>
        <wire>
                <target>parser-impl</target>
                <source>parser-export</source>
        </wire>
</composite>
```

This publishes a component that is able to interpret xml elements and construct an object model that represents the binding. Here is a simple example of a binding xml parser:

```
public class MyBindingParser extends CompoundDescriptorParser {
    public QName getName() {
        return new Qname("binding.example");
    }
    public void parse(XMLStreamReader reader, DescriptorParsingContext
context) throws XMLStreamException {
        CompoundDescriptor parent = (CompoundDescriptor) context.peek();
        BindingDetails fragment;
        if (parent instanceof Service) {
            fragment = new
BindingDetails("com.example.binding.service");
        }
        else if (parent instanceof Reference) {
            fragment = new
BindingDetails("com.example.binding.reference");
        }
        else {
            throw new XMLStreamException("unexpected parent: " +
parent.getClass().getCanonicalName());
        }
        if (parent.accept(fragment)) {
            context.push(fragment);
            parseCompound(reader, context);
            context.pop();
        }
```

```
        else {
            throw new XMLStreamException("Parent does not accept " +
getName(), reader.getLocation());
        }
    }
}
```

When this service is installed when an xml element is found that matches <binding.example /> the parse element will be called. The parser creates a different binding implementation depending on the parent element that encloses it. For example if the composite contained:

```
<service name="foo">
        <binding.example />
</service>
```

then the binding details created would be new BindingDetails("com.example.binding.service");

On it's own this example is fairly limited but it is possible to extend the parsing to attach org.cauldron.newton.descriptor.component.impl.BindingFacet 's to BindingDetails which can contain specific details about the binding. For example if we extended the element to contain an async attribute: <binding.example asyc="true" /> then the parser could attach a BindingFacet that encapsulates this info which can then be parsed to the relevant binding factory at deployment time.

Model To Implementation

Having parsed the xml and built up a BindingDetails object with associated BindingFacets the installer parses the model onto the CompositeFactory. This delegates creation of bindings to services discovered in the osgi registry which implement the ReferenceBindingFactory and ServiceBindingFactory.

These services each provide a method getName(). The composite factory uses this name property to match the factory to the name of the BindingDetails object for a particular binding.

Therefore to continue our example to implement the ExampleReferenceBindingFactory we would first create a composite like so:

```
<composite name="binding-factory">
        <service name="factory-export">
                <interface.java
interface="org.cauldron.newton.component.factory.api.ReferenceBindingFactory"
/>
                <binding.osgi />
        </service>
        <component name="factory-impl">
                <implementation.java.callback
impl="com.example.binding.MyBindingFactory" />
        </component>
        <wire>
                <target>factory-impl</target>
                <source>factory-export</source>
```

```
        </wire>
</composite>
```

This publishes our binding factory to the osgi registry where it will be discovered by the composite factory and called when an composite is installed with a BindingDetails object that matches the name of our binding factory.

Here is a simple example of how the factory could be implemented:

```
public class MyBindingFactory implements ReferenceBindingFactory {
    public MyBindingFactory() {
    }
    public String getName() {
        return "com.example.binding.reference";
    }
    public ReferenceBinding getReferenceBinding(Reference ref,
            BindingDetails provider, ComponentContext ctx)
            throws CompositeFactoryException {
        return new MyReferenceBinding()
    }
    public void releaseReferenceBinding(ReferenceBinding binding) {
    }
}
```

This binding factory creates a new MyReferenceBinding which will handle life-cycle events from the outside world and pass them onto the Newton framework.

2.4.1.3. Creating a Reference Binding

The reference binding interface provides the following methods:

```
public interface ReferenceBinding {
        String getName();
        Reference getReference();
        void startBinding();
        void stopBinding();
}
```

The first method getName() should match the name of the binding factory that created it, i.e. in the example given above "com.example.binding.reference".

The second method getReference() returns an object that implements the Reference interface. It is recommended that impementors simply return a ReferenceImpl for the current release though this may be improved in future releases based on feedback.

The final two methods start/stop binding are used to signal to the binding that it should make/drop external connections to the implemention service layer.

When the service layer detects that the service that is bound to this reference is found/lost it should call the methods

```
void connect(Object serviceObject, Map<String, Object> attrs);
void disconnect(Object serviceObject, Map<String, Object> attrs);
```

On the Reference object returned from getReference() this informs the Newton infrastructure that a service implementation matching the criteria in the reference has been found or is lost.

2.4.1.4. Creating a Service Binding

A service binding provides the following methods:

```
public interface ServiceBinding {
        String getName();
        Service getService();
        void publish();
        void unpublish();
        void onCompositeBound();
        void onCompositeUnbound();
}
```

The first method getName() should match the name of the binding factory that created it, i.e. in the example given above "com.example.binding.service".

The second method getService() returns an object that implements the Service interface. It is recommended that implementors simply return a ServiceImpl for the current release though this may be improved in future releases based on feedback.

The following methods publish/unpublish and onComposite Bound/Unbound have similar semantics but may be used by implementing bindings in different ways.

Publish/Unpublish are called at the beginning and end of a service's lifecycle, they are called once and can be used to set up any long running resources.

onComposite Bound/Unbound is called as references are bound to the underlying component.

These two different lifecycles are useful if your service layer is able to differentiate between “available” and “active”. In the “available” case publish is called but not setCompositeBound – this means semantically “here is a service, though it's not fully functional yet”. In the active case this means the service is visible and all it's dependencies are satisfied so this means “here is a service, come us it”

2.4.1.5. Jars & API's

The main jars that are of interest to the binding developer are found in the directory sdk/lib of a Newton install, they are:

model.jar – contains classes that represent the model of components, services and references

descriptor-parsing.jar – contains classes that parse xml documents into the model

component-api.jar – contains classes that represent the runtime view of components, services and references

component-factory-api.jar – contains interface classes for building extension service/reference bindings or component implementations.

2.5. Content Distribution Service

2.5.1. Overview

The CDS (Content Distribution Service) is used to share binary content between separate Newton processes. The initial use that end-users will see CDS put to is to store OSGi bundles so that they can be installed in any Newton container in the Fabric.

Content is stored in CDS under a specific name and an arbitrary set of attributes and values. For example a bundle foo with version 1.0 and provider codecauldron.org could be stored in cds as:

```
foo,version=1.0,provider=codecauldron.org
```

Currently content within CDS is stored in various *zones*. The default configuration supports the idea of a boot and a remote zone: content stored in the boot zone is accessible from the file system local to a Newton process; content stored in the remote zone is stored in a remote file system and is made available to other Newton processes via a proprietary internal protocol.

Content within the CDS may be accessed either via a programatic interface CDService, or via URL's. The CDS is backed by two sub services IndexService and StorageService. IndexService stores light weight (i.e. small amount of binary data) references to content items stored in CDS. StorageService provides the underlying physical storage of content to which the index items point.

2.5.2. CDService API

The CDService is published to the local osgi registry of the Newton process. It can be used to programattically access and manipulate content within the repository.

```
package org.cauldron.newton.cds.content.api;
import java.io.IOException;
import java.util.Map;
import java.util.Set;
import org.cauldron.newton.ldap.model.LDAPExpr;
/**
 * The content distribution service is used to distribute content around
a distributed system This may include
 * heavyweight files such as large jars. Lightweight metadata is not
propagated using the CDS, but rather the index
 * service. This includes cds entry metadata. This is so that the
resource view is updated out of band of large file
 * transfers, improving its consistency.
 *
 * Content entries are uniquely identified a name, some name-value
attibutes and associated content bytes.
 *
 * Content entries exist in a zone. The range over which updates
propagate is determined by the zone. For example, the
 * 'boot' zone includes only a single vm, and includes resources
necessary to boot the container, and which must be
```

```
 * available before the vm has joined the distributed CDS.
 *
 * Content entries do not have the content bytes embedded - they just
provide access to streams for reading and writing
 * content.
 *
 */
public interface CDService {
    /**
     * The zone which this cdservice provides content for
     */
    String getZone();
    /**
     *
     * @param name
     *            content name - null is wild
     * @param query
     *            ldap style query over identity attributes
     * @return set of matching entries
     * @throws IOException
     */
    Set<Content> findContent(String name, LDAPExpr query) throws
IOException;
    /**
     * look up a specific piece of content by specifying its name and
identity attributes exactly. If there is no match
     * a new Content object with these parameters is created. However it
is not persisted the the CDS until some content
     * bytes are written to it.
     *
     * @param name
     * @param exactMatchAttributes
     * @return the single matching entry.
     * @throws IOException
     */
    Content getContent(String name, Map<String, String>
exactMatchAttributes) throws IOException;
    /**
     *
     * @param url
     * @return A content entry matching this url
     * @throws IOException
     */
    Content urlToContent(String url) throws IOException;
    /**
     * removes all content entries with the given name (null wild) and
matching the query
     *
     * @param name
     * @param query
     * @throws IOException
     */
    void deleteAll(String name, LDAPExpr query) throws IOException;
    /**
     * adds a listener for changes to content matching the supplied name
and filter
     *
     * @param listener
     * @param name
```

```
     * @param filter
     */
    void addContentListener(ContentListener listener, String name,
LDAPExpr filter);
    /**
     * removes a listener for changes to content matching the supplied
name and filter
     *
     * @param listener
     */
    void removeContentListener(ContentListener listener);
}
```

2.5.3. CDS URL Syntax

Alternatively within a Newton process you can access content from CDS via `java.net.URL`'s of the form:

```
cds://contentname?zone=zname&attr1=val1&attr2=val2
```

This allows the developer to read content from cds but not to write or delete it. To write or delete content you must use the CDService API.

2.5.4. Future Work

The zone concept is a temporary pattern we've used for the time being but in and ideal world we'd like to get rid of it. It's purpose is to specify where content is saved when it is written to cds.

However knowing where the content is stored is actually non beneficial to a range of patterns. For example it would be useful to access content via url in a more “natural” form: cds://name?attr=val vs having the artificial zone= attribute tagged onto the name. This trips us up when we come to load content into cds that has dependent files as all the dependent files need to be updated with the zone on write (not very elegant).

Problems with removing this come when you consider issues of consistency across multiple repositories, i.e. if content is stored in two areas, local file system and remote file system which content is being read/written to for a given set of name attributes?

Anyone with answers or interested in further discussion is encouraged to contact us via our dev (http://newton.codecauldron.org/mailman/listinfo/dev) mailing list.

3. Examples

3.1. Examples: Overview

In order to run the examples you need to install Newton.

The examples are already compiled in the distribution. If you want to rebuild the examples, use Apache Ant as follows:

```
cd examples
```

```
ant
```

Note:
The examples are actually built using Sigil (http://sigil.codecauldron.org) and Apache Ivy.

The following examples take you from the basic installation and dynamic wireup of composites within a single Newton container, right through to the use of Systems to dynamically provision and wire up graphs of components spanning multiple containers.

Chainlink
This is the example to try first. Several link composites are installed, dynamically wired up and rewired in response to a change. Then everything is uninstalled leaving Newton in its initial state.

Remote Chainlink
Repeats the chainlink demo, showing how Newton uses Jini to distribute the chain across several JVMs.

Spring OSGi Chainlink
In this example we show how Newton can be used to manage Spring OSGi bundles.

Scatter Gather
Shows how to build a scatter-gather grid using Newton. This example makes use of a Javaspace.

Scatter Gather System
Repeats the last demo using Environments to deploy and dynamically scale the application.

Smart Proxy
Shows how Jini Smart Proxies are supported by Newton.

Fractal Render
Combines concepts used in previous demos to show how to build a real world application to render fractal images using a scatter gather pattern.

3.2. Chainlink Demo

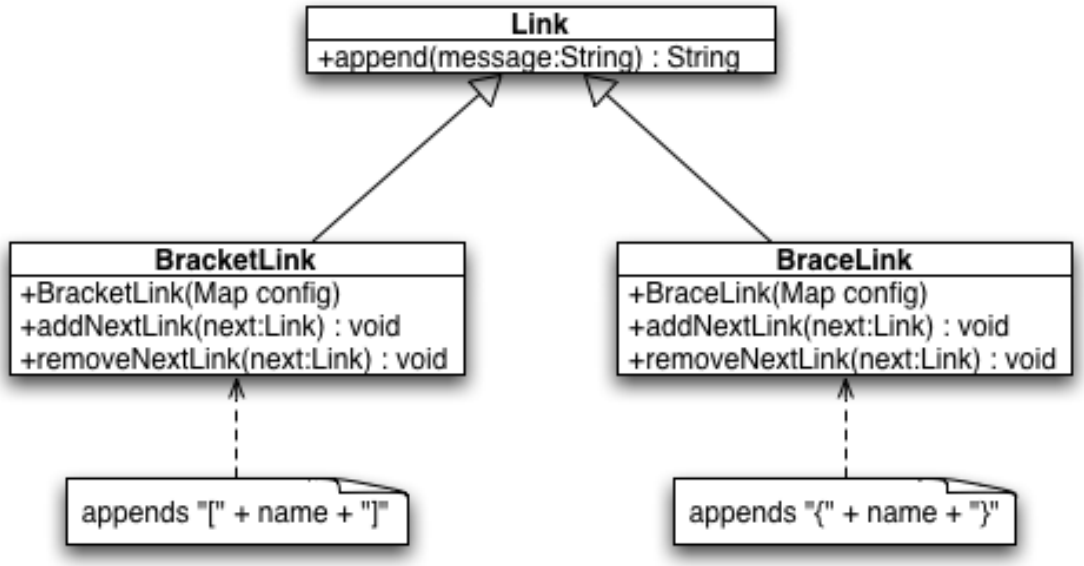
3.2.1. Summary

The chainlink demo application provides an introduction to Newton's SCA based component model. The demo runs entirely within a single JVM. The Remote Chainlink Demo extends this to the multi-JVM case.

The aim is to illustrate the full lifecycle of a deployable Newton composite, starting with dynamic resolution of install-time resources, then examining runtime auto-wireup of service dependencies, and finally looking at garbage collection of unused resources after all chainlink components have been removed.

3.2.2. Overview

Chainlink builds a chain out of Newton composites implementing the `Link` interface. `Link` has one method, `String append(String txt)`. Implementations append some text to the String they are passed and then forward it on to the next link if there is one, otherwise returning the current value of the String. The demo makes use of two composite types that export services implementing the `Link` interface. Brace-link composites append their link name in braces, i.e "{name}", whereas Bracket-link composites use brackets, i.e. "[name]". The class structure is illustrated below.



At the start of the demo, the Java classes which define the composites and the `Link` interface they implement are unknown to the Newton container. All are loaded dynamically as the composites are installed.

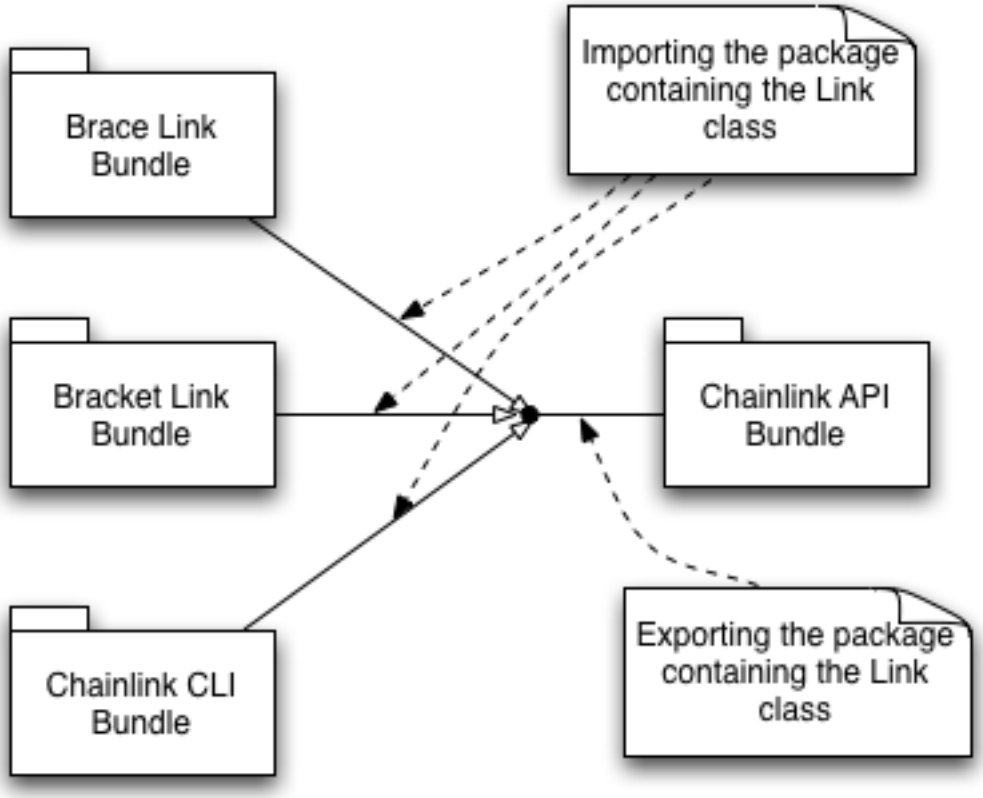
In Newton, the job of creating composites is delegated to composite specific factory bundles. These supply the code needed to create the composites, and act as factories for turning composite instance descriptors into runtime composite instances.

We will be creating chains that using a mixture of the two link composite types. To avoid class mismatch problems when we wire up the composites we must ensure that they both see the same version of the `Link` interface through which they connect. This means ensuring that the version of `Link` each uses is defined by the same ClassLoader. In OSGi this can be achieved by having both factory bundles import the package containing the `Link` interface from a third bundle, which we'll call the api bundle. See OSGi classloading for more details.

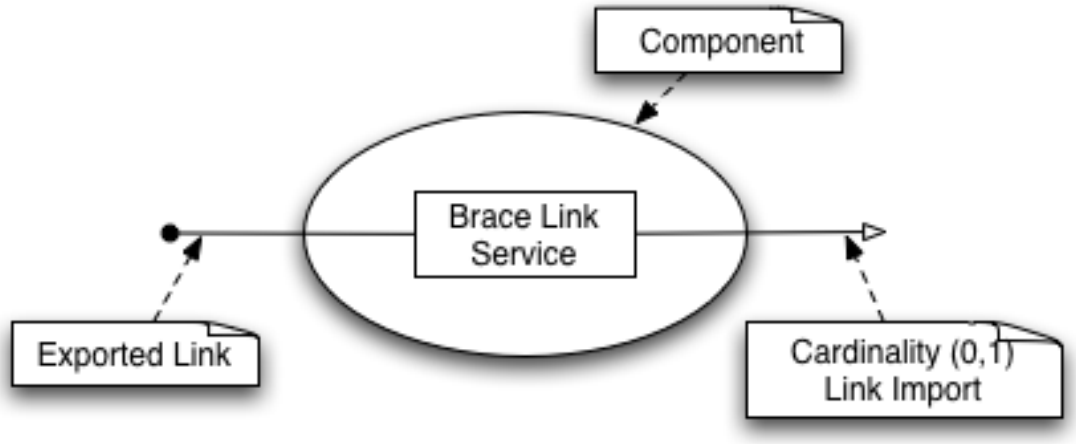
The above differs from the situation in traditional J2EE application servers, where extensibility and dynamic replaceability of components is limited to those implementing one of the fixed set of interfaces pre-defined by the application server, e.g. the Servlet interface. Newton makes use of OSGi's powerful peer classloading model to provide these facilities for arbitrary interfaces.

As well as links in the chain we also make use of a Command line interface (CLI), this has its own factory bundle, and also imports the link interface from the api bundle.

The bundles and their dependencies are illustrated below:



Each factory bundle contains a composite template that describes the services the composite provides and the external references it uses. The two link composites provide a service implementing the `Link` interface and try to connect to references implementing the same interface. The chain will built by dynamically wiring up services and references corresponding to the different Link interfaces. The composite structure defined by the brace-link template is illustrated below. The bracket-link template is essentially the same.



Here the component is based on an instance of the ExportingBraceLink class. It provides a service called link, which is obtained by calling the instance's _exportLink() method The multiplicity of (0,1) on the reference means that the composite wants to connect to at most 1 `Link` interface, but can still function if it can't find any. The wires link the service and reference to the concrete component.

The XML for this template is shown below. See The Newton Component Model for a walkthrough of composite template syntax.

```
<?xml version="1.0"?>
<composite name="braceLinkTemplate">
    <description>chainlink demo brace link</description>

    <reference name="nextLink" multiplicity="0..1">
        <interface.java
interface="org.cauldron.newton.example.chainlink.interfaces.Link"/>
        <binding.osgi/>
    </reference>

    <service name="brace-link-export">
        <interface.java
interface="org.cauldron.newton.example.chainlink.interfaces.Link"/>
            <binding.osgi/>
    </service>

    <component name="brace-link">
        <description>A brace decorated chain link</description>
        <property name="linkName" value="Unnamed" type="string"/>
        <service name="link"/>
        <reference name="nextLink"/>
        <implementation.java.callback
impl="org.cauldron.newton.example.chainlink.brace.ExportingBraceLink"/>
    </component>

    <wire>
        <source.uri>brace-link/nextLink</source.uri>
        <target.uri>nextLink</target.uri>
```

```
    </wire>

    <wire>
        <source.uri>brace-link-export</source.uri>
        <target.uri>brace-link</target.uri>
    </wire>
</composite>
```

3.2.3. Running the demo

3.2.3.1. Booting the Newton container

In order to run the demo it is first necessary to start a single Newton container. To do this change to the bin directory and run

```
//for unix
$ ./container

//for windows
> container.bat
```

After starting Newton wait for the "Boot complete" message, after which Newton makes a command line available.

3.2.3.2. Loading bundles into CDS

The Content Distribution System (CDS) is Newton's distributed repository. At this point the bundles used by the chainlink demo are not present in the CDS, so they must be loaded. Note that this doesn't install anything, it just makes resources available for future installations.

To load the demo bundles run:

```
> cds scan boot examples/chainlink/build/lib
```

This loads all of the bundles in demo/build/lib into the *boot zone* of CDS, extracting metadata and indexing the bundles as they are loaded. Resources in the boot zone are local to the machine on which they are loaded.

3.2.3.3. Installing Chainlink instances

We now install several `Link` composites, all of type brace-link. Each composite is instantiated using a composite instance descriptor that identifies its factory bundle and constrains its template. As the first composite of type brace-link is being installed Newton looks up its factory bundle, resolves all of the bundles this depends on, and installs them. The factory bundle itself is then installed and started, after which it serves to convert instance descriptors into runtime composite instances.

To install the composites run the following.

```
> installer install
examples/chainlink/build/etc/chainlink-local-a.composite
> installer install
```

```
examples/chainlink/build/etc/chainlink-local-b1.composite
> installer install
examples/chainlink/build/etc/chainlink-local-c.composite
```

To see the newly installed bundles enter `bundles -i` at the Newton console. To get an OSGi level view of the newly installed services enter `services -i -l`.

The three composite instances constrain the template metadata in different ways:

- Composite A annotates its exported Link interface with tag=a and filters its imported a Link interface using tag=b
- Composite B annotates its exported Link interface with tag=b and filters its imported a Link interface using tag=c
- Composite C annotates its exported Link interface with tag=c and filters its imported a Link interface using tag=d

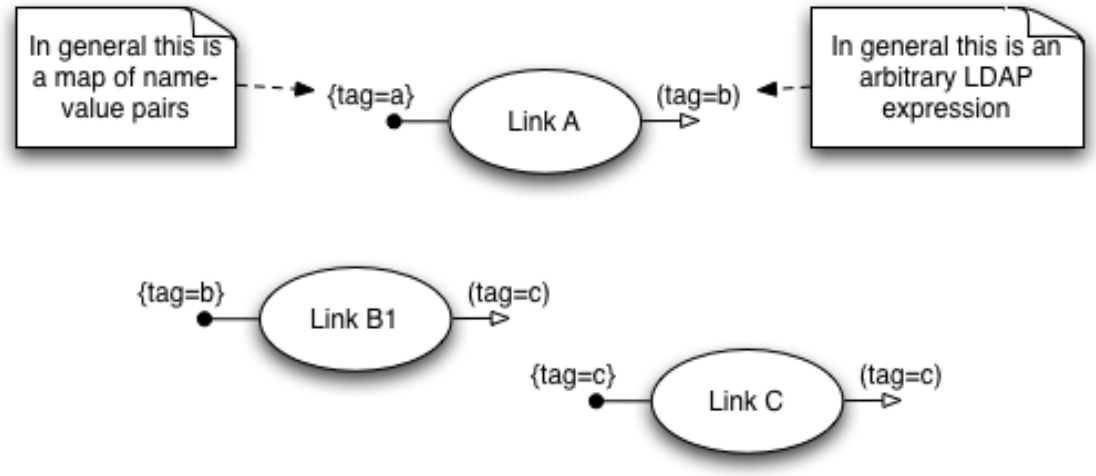
The instance descriptors for the three components are much the same, as an example here's the instance descriptor XML for component A.

```
<?xml version="1.0"?>
<composite name="LinkA">

    <bundle.root bundle="chainlink-local-brace-bundle" version="1.0"/>
    <include name="braceLinkTemplate"/>

    <reference name="nextLink">
        <binding.osgi filter="(tag=b)"/>
    </reference>

    <component name="brace-link">
        <description>link A</description>
        <property name="linkName" value="A-link" type="string"/>
    </component>

    <service name="brace-link-export">
        <binding.osgi>
            <attribute name="tag" value="a" type="string"/>
        </binding.osgi>
    </service>

</composite>
```

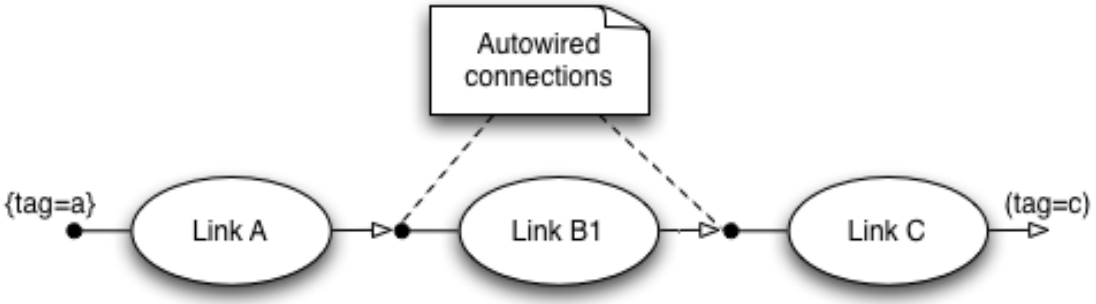
The <bundle.root ..> tag identifies the factory bundle for this composite type. The next line imports a composite template from this bundle, and the rest of the document overrides settings in this template. The <binding.osgi ..> elements define the scope of the services and references, and the <property ..> entries are passed to the service class constructor as a map. For more information see The Newton Component Model.

In general services are annotated using a name-value attribute map, and references are filtered using LDAP syntax.

Diagrammatically we have:



As the composites are installed Newton examines their external interfaces, attributes and filters, and uses these to organize them into a chain as below



3.2.3.4. Installing the chainlink Command line interface.

At this point the three brace-link composites have been installed and automatically wired up into a chain. In order to make calls on the chain we are going to a install a chainlink CLI composite. This composite has the following structure.

To install the CLI type:

```
> installer install
examples/chainlink/build/etc/chainlink-local-cli-a.composite
```

Once installed it's `Link` reference looks for a `Link` service satisfying it's (tag=a) filter. This causes it to bind to Link A. It's `ConsoleCommandHandler` service is dynamically wired up to a reference exposed by Newton's command line infrastrcuture and used to add a new command group to the Newton command line. To see the new command type:

```
> help
Available command groups (type 'enter' to enter a group):
system - commands for manipulating systems
storagecds - Commands for manipulating cds storage
sleep - sleeps for a configurable number of milliseconds
session - Session commands built into the console
provisioner - Commands for manipulating the provisioner
obr - OBR commands
logman - Commands for controlling logging within container
logconfig - Configuration commands for the log.
log - Log commands
installer - installs/uninstalls/tracks SCA components in this jvm
indexcds - Commands for manipulating the index service
ifconfig - Tools for managing configuration providers
framework - Framework commands
exec - executes a script from a file or url location
configuration - Configuration commands
chainlink-a - Commands for manipulating chains of Links
cds - Commands for manipulating cds content
binder - Tools for introspecting the binding graph
```

The command group we've just added is chainlink-a (third from bottom).

We can now use this command to make calls on the chain. When we do this each link appends its name in braces to the string it is passed until the end of the chain is reached. This is then returned along the chain to the original caller. Invoke the chain as follows:

```
> chainlink-a chain
chain: cli->{A-link}->{B1-link}->{C-link} [end of chain]
```

3.2.3.5. Dynamically rewiring the chain.

So far we've gone from a blank container to a wired up set of brace-link composites. We will now dynamically swap out the middle link, replacing it with a bracket-link composite.

First lets install the B2 composite. Since composite A's Link reference is already satisfied by the Link service provided by composite B1, composite B2's Link service is initially unused.

```
> installer install
examples/chainlink/build/etc/chainlink-local-b2.composite
```

You can run `chainlink-a chain` again to verify that the chain hasn't changed. You can check that the bracket link bundle has loaded using `bundles -i` and that the new composite's Link interface has been exported using `services -i -l`

We now uninstall the B1 composite

```
> installer uninstall
```

```
examples/chainlink/build/etc/chainlink-local-b1.composite
```

When B1 is removed, Newton realizes that composite A's Link reference is no longer satisfied, identifies the Link service on B2 as a suitable replacement and dynamically rewires the chain as we can see by running:

```
> chainlink-a chain
chain: cli->{A-link}->[B2-link]->{C-link} [end of chain]
```

Note that this time the name "B2-Link" is in brackets rather than braces.

3.2.3.6. Uninstalling chainlink & bundle garbage collection

Before uninstalling the chain take a look at the installed bundles by running:

```
> bundles -i
   id  level/state name
   --------------------
   ... (first 71 omitted) ...

   72  1/active    org.cauldron.newton.provisioner.service
   73  1/active    org.cauldron.newton.provisioner.client
   74  1/resolved  com.sun.jmx.remote.optional
   75  1/active    org.cauldron.newton.management.server
   76  1/active    org.cauldron.newton.management.monitor
   77  1/active    org.cauldron.newton.boot.control
   78  1/resolved  chainlink-api-bundle
   79  1/active    chainlink-local-brace-bundle
   80  1/active    chainlink-local-bracket-bundle
   81  1/active    chainlink-local-cli-bundle
```

Bundles 78 to 81 are the ones that have been installed to support the chainlink composites.

Next uninstall all of the chainlink composites:

```
> installer uninstall
examples/chainlink/build/etc/chainlink-local-a.composite
> installer uninstall
examples/chainlink/build/etc/chainlink-local-b2.composite
> installer uninstall
examples/chainlink/build/etc/chainlink-local-c.composite
> installer uninstall
examples/chainlink/build/etc/chainlink-local-cli-a.composite
```

The components are now uninstalled. Shortly after this the bundle garbage collection mechanism, which runs on a bckground thread, will notice that the chainlink bundles are unused and remove them. You can speed this up by running

```
>installer gc
```

which will schedule a garbage collection sweep in the next 10 seconds. After this you can check that all unused bundles have been removed:

```
> bundles -i
   id  level/state name
   --------------------
   ... (first 71 omitted) ...
```

```
  72  1/active    org.cauldron.newton.provisioner.service
  73  1/active    org.cauldron.newton.provisioner.client
  74  1/resolved  com.sun.jmx.remote.optional
  75  1/active    org.cauldron.newton.management.server
  76  1/active    org.cauldron.newton.management.monitor
  77  1/active    org.cauldron.newton.boot.control
```

At this point we are right back where we started, and have a blank container that knows nothing about chainlink or any of its classes. (Note that the chainlink bundles are still available in CDS, but these can easily be removed too, see CDS, although in a production system the contents of the distributed CDS would typically be long lived.)

3.2.4. Further exercises

1. Observe the dependency and garbage collection code operating by installing/uninstalling composite instances in different orders.
2. Using a combination of multiplicities, tags and filters it should be possible to create any number of chainlink configurations. Note some configurations may require modifications to the simple command line or link code if they specify circular dependencies (Newton copes well with circular dependencies, but individual applications may not)

3.2.5. Follow on Demos

1. The Remote Chainlink Demo extends the current example to the multi JVM case

3.3. Remote Chainlink Demo

3.3.1. Summary

This demo extends the Chainlink Demo so that it runs across several JVMs, and uses this to illustrate Newton's remoting capabilities.

The aim is show how to bootstrap a distributed Newton fabric, and to illustrate how composites deployed in different containers can be dynamically wired up and rewired in response to changes.

Newton remoting makes use of RMI technology. Newton isolates end users from many RMI related concerns. The remaining considerations are highlighted for developers in the following example.

3.3.2. Overview

The structure of the example is almost identical to that for the single JVM Chainlink Demo, so rather than repeat that we'll focus here on the differences.

The bundles used in this demo are analogous to those used in the local demo. The differences are as follows:

1) The bundle names have the form chainlink-rmi-brace-bundle.jar rather than chainlink-brace-bundle.jar.
2) Descriptor names have the form chainlink-rmi-a.xml rather than chainlink-local-a.xml
3) The remote versions of the brace-link, bracket-link and api bundles contain an RMI-Codebase manifest header. This identifies the code needed to remotely reconstitute the service proxies that are remotely exported by the example.

A publish.xml file is also generated. Publish files are read by Newton as their containing bundles are loaded into CDS. They identify nested content that needs to be added to CDS in its own right. In the current case we are using it to publish the RMI codebase jar to CDS, from where it can be accessed by service clients. Here's the publish.xml file:

```
<?xml version="1.0"?>
<publish>
    <item name="chainlink-brace-dl.jar" path="chainlink-brace-dl.jar">
        <attribute name="type" value="rmi.codebase"/>
    </item>
</publish>
```

Note:

The Sigil build tool automatically generates the RMI-Codebase header and publish.xml file, from the -composites directive in sigil.properties.

See RMI in Newton for more information.

4) In the remote version services and references are exposed by RMI rather than via the

OSGi registry scope. Here's the Remote version of the brace-link template.

```
<?xml version="1.0"?>
<composite name="rmiBraceLinkTemplate">
    <description>chainlink rmi demo brace link</description>

    <reference name="nextLink" multiplicity="0..1">
        <interface.java
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link"/>
        <binding.rmi />
    </reference>

    <service name="brace-link-export">
        <interface.java
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link"/>
        <binding.rmi />
    </service>

    <component name="brace-link">
        <description>A brace decorated chain link</description>
        <property name="linkName" value="Unnamed" type="string"/>
        <reference name="nextLink"/>
        <implementation.java.callback
impl="org.cauldron.newton.test.bundles.chainlink.brace.BraceLink"/>
    </component>

    <wire>
        <source.uri>brace-link/nextLink</source.uri>
        <target.uri>nextLink</target.uri>
    </wire>

    <wire>
        <source.uri>brace-link-export</source.uri>
        <target.uri>brace-link</target.uri>
    </wire>
</composite>
```

Note the <binding.rmi ..> elements. The fabric name comes from the container.fabricName system property if this is set, and the user.name property if it is not. This value can be overridden by using a group attribute on the binding.rmi tag. Note that in order for a number of remote services to see each other they should all be using the same fabric name. The standard way to do this in newton is by setting container.fabricName. The scripts which start newton propagate the platform level environment variable NEWTON_FABRIC_NAME to the java system property container.fabricName. The property can also be set on containers as they are started by using the -fabricName flag

The remote chain CLI is interesting because has an RMI scoped reference to a `Link`, but provides an OSGi scoped `ConsoleCommandHandler` service. This is because we want to use the command in the console it is launched in! Here's the template XML

```
<?xml version="1.0"?>
<composite name="rmiChainlinkCli">
    <description>chainlink rmi cli</description>

    <!-- remote, rmi scoped import -->
```

```
    <reference name="connection" multiplicity="0..1">
        <interface.java
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link"/>
        <binding.rmi/>
    </reference>

    <!--  local, OSGi scoped export -->
    <service name="cli-export">
        <interface.java
interface="org.cauldron.newton.command.console.ConsoleCommandHandler"/>
        <binding.osgi/>
    </service>

    <component name="cli">
        <description>cli for the chainlink demo</description>
        <property name="contextName" value="chainlink" type="string"/>
        <reference name="connection"/>
        <implementation.java.callback
impl="org.cauldron.newton.test.bundles.chainlink.cli.Commands"/>
    </component>

    <wire>
        <source.uri>cli/connection</source.uri>
        <target.uri>connection</target.uri>
    </wire>

    <wire>
        <source.uri>cli-export</source.uri>
        <target.uri>cli</target.uri>
    </wire>
</composite>
```

3.3.3. Running the example.

3.3.3.1. Prerequisites

Try the local version of the Chainlink Demo before this multi-JVM one, without it this demo may lack context.

3.3.3.2. Booting the Newton container

In order to run the demo it is first necessary to start two Newton containers. In order to see each other these need to be running as part of the same fabric. To start the two containers on a single machine open two terminals in the Newton bin directory and run

In terminal 1

```
$ bin/container -fabricName=myfabric -instance=1
```

In terminal 2

```
$ bin/container -fabricName=myfabric -instance=2
```

Note that this demo will also work when running across several machines (firewalls permitting)

The -fabricName flag sets the fabric name. We've used 'myfabric' but you should choose your own name to prevent clashes. Note that if the fabric name is not set it defaults to the current user name. This means that simple tests on a single machine will work without the fabricName being set. However it should always be set when more than one machine is involved

The -instance flag is used to separate the state belonging to the two conatiner instances. This is only necessary because we are running the two instances out of the same Newton install. If the instances were running on different machines there would be no need for this flag

After starting Newton wait for the "Boot complete" message, after which Newton makes a command line available.

3.3.3.3. Booting the distributed runtime

Newton's distributed infrastructure makes use of multicast, so it is essential that your network and computers are in a configuration that allows multicast traffic. If you are unsure of your setup, or having difficulty running this example the please see Troubleshooting Multicast.

Every instance of Newton starts up with the client side of the distributed infrastructure already installed. At present it is necessary to manually boot the server side infrastructure. To do this enter the following at one of the Newton command lines

```
> exec etc/scripts/single-jvm-dist-infra
```

This installs Newtons remote registry, distributed repository, and other distributed runtime components

This remote infrastructure is itself implemented as an SCA system built from Newton composites.

Optionally you can start the jinibrowser composite, a GUI tool for viewing the remote registry. You can use this to see the services that available to your group remotely.

```
>installer install etc/instances/jinibrowser.composite
```

The services may take a few seconds to come up - you'll get an error message at the next step if you are too quick.

3.3.3.4. Loading bundles into CDS

At this point the bundles used by the remote chainlink demo are not present in the CDS repostory, so they must be loaded. Note that this doesn't install anything, it just makes resources available for future composite installations.

To load the demo bundles run the following in one of your Newton instances.

```
> cds scan remote examples/chainlink/build/lib
```

This loads all of the bundles in demo/build/lib into the *remote zone* of CDS, extracting

metadata and indexing the bundles as they are loaded. Resources in the remote zone can be seen by all Newton containers in the same Fabric

3.3.3.5. Creating the Chain

We now create a chain of brace-link composites that is distributed across two JVMs.

In The first Newton container we install the CLI and the first link in the chain as follows.

```
> installer install
examples/chainlink/build/etc/chainlink-rmi-cli-a.composite
> installer install
examples/chainlink/build/etc/chainlink-rmi-a.composite
```

In this remote case the command group added by the CLI is called rmiChain-a, so to see the chain so far you can run

```
> rmiChain-a chain
chain: cli->{A-link} [end of chain]
```

So far this is all in one JVM. To demonstrate the distributed behaviour we create the rest of the chain in the second JVM

```
> installer install
examples/chainlink/build/etc/chainlink-rmi-b1.composite
> installer install
examples/chainlink/build/etc/chainlink-rmi-c.composite
```

then, in the first JVM - i.e. the one you installed the CLI in rerun

```
> rmiChain-a chain
chain: cli->{A-link}->{B1-link}->{C-link} [end of chain]
```

to show the distributed chain structure.

3.3.3.6. Modifying the chain.

So far we've built up a chain of brace-links that spans two Newton containers. Now we'll make add a link in one container and remove one in the other, forcing Newton to rewire the chain.

In container 1 run

```
> installer install
examples/chainlink/build/etc/chainlink-rmi-b2.composite
```

This adds a new 'b' composite to the container.

And in container 2 run

```
> installer uninstall
examples/chainlink/build/etc/chainlink-rmi-b1.composite
```

This removes the 'b' composite in container 2. When this happens Newton notices that link a has unbound references and rewires it to the 'b' link in the other container. To see the result run the following in the container you added the CLI to.

```
> rmiChain-a chain
chain: cli->{A-link}->[B2-link]->{C-link} [end of chain]
```

Here brace-link B1 which was in container 2 has been replaced by bracket-link B2 in container 1.

3.3.4. Further exercises

Apart from the CLI the different parts of this demo could have been run in either container, or in any additional containers started with the same fabric name.

Try installing links and infrastructure composites in different containers and on different machines. For convenience you can install the CLI in as many containers as you like.

As with the local chainlink demo the bundles that implement these composites will be removed by the Newton's garbage collection mechanism as the composites that need them are removed.

Hard kills of Newton containers may take a little while to spot, although it will ultimately react to the change.

3.4. Spring OSGi based chainlink demo

3.4.1. Introduction

3.4.1.1. Overview

In this example we show how Newton can be used to manage Spring OSGi bundles. The specific features demonstrated are:

1. Deployment and full lifecycle management of Spring OSGi bundles using Newton
2. Automatic installation of the Spring-OSGi runtime whenever Spring-OSGi bundles are in use, and its automatic removal when it is no longer needed
3. How Spring-OSGi bundles are interpreted as SCA composites by Newton
4. How to configure Spring-OSGi bundles using SCA properties
5. How to promote OSGi level services and references to SCA level so that they can work non OSGi bindings to, for example, communicate remotely.
6. Use of Newton's provisioning infrastructure to deploy and maintain a distributed system based on Spring OSGi bundles.

Note:

This example (examples/spring) uses Spring Maven archetypes with minimal modification. There is also a similar example (examples/spring.chain) that uses the Sigil OSGi build tools that are used by the other examples.

3.4.1.2. Prerequisites

To build and run the example you will need the following:

- Newton version 1.3 or higher
- The Spring-OSGi for Newton example (examples/spring in Newton distribution)
- Java 5
- Spring-OSGi version 1.1.2 (http://www.springframework.org/osgi) (with dependencies)
- Ant 1.7 0 or later (http://ant.apache.org/bindownload.cgi)
- Maven 2.0 (http://maven.apache.org/download.html)
- At least one OS instance (Linux, OS X or Windows). If more OS instances are available to you, you can use them as well.

Maven, Ant, Spring OSGi, and the examples need only be installed on your development machine, but Newton must be installed on any OS instances that you use to run these examples.

3.4.1.3. Developing Spring-OSGi bundles

This is not a tutorial on general Spring-OSGi bundle development. The following links may be useful to you if you are not already familiar with Spring-OSGi

- Spring-OSGi documentation (http://static.springframework.org/osgi/docs/1.0.1/reference/html/)
- Spring-OSGi Maven archetype (http://www.springframework.org/node/361) . This archetype was used to create all of the demo bundles.

3.4.2. Installing and preparing the examples

The examples are installed under examples/spring in the binary Newton distribution.

If you install them somewhere else, you must set the NEWTON_HOME environment variable to point at the Newton install directory.

Your local Maven repository also needs to be populated with some Newton artefacts. To do this enter the examples/spring directory and run.

```
ant prepare.examples
```

This only has to be done once, no matter how many times you build the examples.

3.4.2.1. Using Eclipse

Eclipse projects for the two demo bundles are supplied with the examples. Before importing the projects add a classpath variable called NEWTON_HOME to your eclipse workspace. This should be set to your Newton install directory. The form for manipulating classpath variables can be found in the eclipse preferences under Java/Build Path/Classpath Variables

To import the demo projects go to File/Import/Existing Projects into Workspace, select the directory where you installed the examples, choose the org.cauldron.examples.spring.chain.cli and org.cauldron.examples.spring.chain.link projects, and press Finish.

3.4.3. The demo application

3.4.3.1. Overview

This example builds a chain made up of links. Each link provides a Link service with a single method *append*, and has a reference to another Link service. Properties and filters are used to control the order in which the links wire up.

As calls pass through the chain each link appends is name, building up a string that shows the full path. When the end of the chain is reached the string is returned.

There are two Spring-OSGi bundles in this demo. One, called springlink, implements the links in the chain. The other, called springchaincli provides an extension to the Newton console that makes it possible to invoke the chain from the command line.

3.4.3.2. Structure

There are two main parts to this example.

1. Local example. In this section we show how the addition of a small amount of metadata can be used to turn Spring-OSGi bundles into SCA components and how SCA properties can be used to configure Spring-OSGi bundles.
2. Distributed example. In this section we show how to promote OSGi level services and references to SCA level so that they can, for example, interact remotely. We also show how Newton's provisioning capabilities can be used to deploy and maintain a distributed Spring-OSGi application.

The bundles in the two parts differ only in their metadata (stored below src/main/resources directory of each Spring-OSGi project). Ant targets for switching to the metadata for each part are supplied.

3.4.4. Local Example

In this initial example we'll show how to deploy and undeploy the springchaincli and springlink bundles to a single Newton container, walking through the necessary Newton specific configuration.

First of all make sure you are using the local example metadata by running the following in the examples directory.

```
ant use.local.chain
```

This copies the local chain metadata into the bundle resources directory (src/main/resources in each Spring-OSGi project). If you are using an IDE, you should refresh your projects after running this task.

Now build the example:

```
ant clean.examples install.examples
```

Before running anything let's look at the configuration files. These are below the src/main/resources directory of each Spring-OSGi project and are organized as follows

```
`-- META-INF
    |-- MANIFEST.MF
    |-- newton
    |   `-- springchaincli.template
    `-- spring
        |-- bundle-context-osgi.xml
        `-- bundle-context.xml
```

Below the spring directory are the standard Spring-OSGi configuration files. bundle-context.xml is just an ordinary spring beans.xml file, while

bundle-context-osgi.xml contains OSGi specific extensions. The newton directory holds a Newton template. This is used to expose the Spring-OSGi bundle as an SCA composite, to configure the bundle and to promote its references and services. The MANIFEST.MF file does not contain the full bundle manifest, most of which is generated by Spring-OSGi's Maven based build system. All it contains are additions to the manifest required by Newton, specifically entries identifying the bundle as a Newton bundle and telling Newton where to look for the template file.

We are now going to look at all of these files in detail. Here they are for springchaincli. Their rather verbose namespace declarations have been omitted for clarity.

bundle-context.xml

```
<beans>
  <bean name="chainLinkCliBean"
class="org.cauldron.newton.examples.spring.chain.cli.impl.ChainLinkCliBeanImpl"
/>
</beans>
```

This contains a single bean. It has no explicit dependencies, but as we shall see from the next file, it has an implicit dependency on a Link (the first link in the chain)

bundle-context-osgi.xml

```
<beans>
  <osgi:reference
    id="nextLink"
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link"
    filter='(tag=a)'
    cardinality="0..1"
  >
  <osgi:listener bind-method="bindLink" unbind-method="unbindLink"
    ref="chainLinkCliBean"/>
  </osgi:reference>
  <osgi:service ref="chainLinkCliBean"
interface="org.cauldron.newton.command.console.ConsoleCommandHandler"/>
</beans>
```

This defines a reference to an OSGi service, implementing Link, with property tag=a. Whenever a service satisfying this reference is found or lost the listener calls back to the chainLinkCliBean bean defined in bundle-context.xml.

Also defined is a service implementing Newton's ConsoleCommandHandler. This provides the command line extension used to interact with the chain.

springchaincli.template

```
<composite name="springchaincli">
  <component name="cli">
    <description>spring chain cli</description>
    <implementation.spring />
  </component>
</composite>
```

This defines an SCA composite, and states that it is implemented by a Spring-OSGi bundle. At this stage all we are interested is the composite name *springchaincli*, which

will be used below.

MANIFEST.MF

```
Manifest-Version: 1.0
Installable-Component: true
Installable-Component-Templates: META-INF/newton/springchaincli.template
```

This adds Newton specific Manifest entries as discussed above.

Here are the equivalent files for springlink

bundle-context.xml

```
<beans>
  <property-placeholder
    persistent-id="org.cauldron.newton.examples.spring.chain.link"/>
  <bean name="springLinkBean"
class="org.cauldron.newton.examples.spring.chain.link.impl.SpringLinkImpl"
>
    <property name="linkName" value="${linkName}"/>
  </bean>
</beans>
```

Again this contains a single bean, but this time it has a property, linkName which is read from the OSGi Configuration Admin service. The property-placeholder line indicates the Configuration Admin *service pid* from which the properties should be read. In the template and instances below we'll see how these can be set using SCA properties

bundle-context-osgi.xml

```
<beans>
  <osgi:reference id="nextLink"
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link"
       filter="(tag=b)" cardinality="0..1">
    <osgi:listener bind-method="bindLink" unbind-method="unbindLink"
ref="springLinkBean"/>
  </osgi:reference>

  <osgi:service ref="springLinkBean"
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link">
    <service-properties>
      <entry key="tag" value="a"/>
    </service-properties>
  </osgi:service>
</beans>
```

This bundle has a reference to an implementation of Link filtered by (tag=b), and provides a service implementing Link with property tag=a. Note that in this local example the (tag=b) filter wont be used, although it will be later on in the in remote example.

springlink.template

```
<composite name="springlink" >
  <component name="springLink">
    <service name="springLink" />
    <description>spring based chain link</description>
    <property name="service.pid"
              value="org.cauldron.newton.examples.spring.chain.link"
```

```
              type="string" />
    <property name="linkName"
              value="Local-A-Default"
              type="string" />
    <implementation.spring />
  </component>
</composite>
```

This template exposes the springlink bundle as a composite and sets its service.pid property. Whenever Newton sees a service.pid property in a component it registers all other SCA properties for that component with the OSGi Configuration Admin service using that service.pid as the key. In this case the service.pid correlates with the persistent-id attribute from the property-placeholder element in bundle-context.xml, so these properties are made available to the Spring configuration files.

MANIFEST.MF

```
Manifest-Version: 1.0
Installable-Component: true
Installable-Component-Templates: META-INF/newton/springlink.template
```

Now build the examples

```
ant clean.examples install.examples
```

The first time you build, Maven may download several libraries required by Spring-OSGi's build system. This may take a few minutes. Subsequent builds will be much faster.

3.4.4.1. Running the local example

To run the demo first start Newton by changing to the Newton bin directory and running

for unix

```
$ ./container
```

for windows:

```
> container.bat
```

If at any time you want to look at Newton's log, you can find it under var/container.log

Once Newton has started we need to load the example bundles and the Spring-OSGi bundles into CDS (Content Distribution System), Newtons repository.

To do this run

```
cds scan boot examples/spring/lib
cds scan boot examples/chainlink/build/lib
cds publish boot examples/spring/xml/publish-springdm-1.1.2.xml
<Spring-OSGi install directory>
```

Here the first line loads the spring example bundles. The second line loads the non-Spring Newton chainlink bundles (from which our Spring bundles import an API package). The third line uses a Newton publish file to load the required bundles from the Spring OSGi install directory. Note that this publish file is specific to version 1.1.2 of Spring-OSGi, so if you are using a later version you should edit it appropriately. All bundles are loaded

into the boot zone of CDS, i.e. they are private to this Newton instance.

Note that CDS takes paths relative to the Newton install directory regardless of the directory from which you launched Newton.

Before installing anything take a look at the bundles that are already installed by running

```
> bundles -i
```

at the Newton command line. This will display all installed bundles, in the order in which they were installed. Here are the last few taken from my machine

```
  82 --/resolved  org.cauldron.newton.presence.local.api
  83 --/resolved  org.cauldron.newton.presence.remote.api
  84 --/active    org.cauldron.newton.presence.service
  85 --/resolved  com.sun.jmx.remote.optional
  86 --/active    org.cauldron.newton.management.server
  87 --/active    org.cauldron.newton.management.monitor
  88 --/resolved  org.cauldron.newton.boot.control.api
  89 --/active    org.cauldron.newton.boot.control
```

Note that no example or spring bundles are present.

It is also worth typing *help* at the console to see the available commands.

```
> help
Available command groups (type 'enter' to enter a group):
system - commands for manipulating systems
storagecds - Commands for manipulating cds storage
sleep - sleeps for a configurable number of milliseconds
session - Session commands built into the console
provisioner - Commands for manipulating the provisioner
obr - OBR commands
logman - Commands for controlling logging within container
logconfig - Configuration commands for the log.
log - Log commands
installer - installs/uninstalls/tracks SCA components in this jvm
indexcds - Commands for manipulating the index service
framework - Framework commands
exec - executes a script from a file or url location
configuration - Configuration commands
cds - Commands for manipulating cds content
binder - Tools for introspecting the binding graph
```

Note that this list does not include any references to our demo.

Now install springchaincli as follows:

```
> installer install composite:springchaincli
```

If we now take another look at the installed bundles using **bundles -i**. the last few bundles should be

```
  88 --/resolved  org.cauldron.newton.boot.control.api
  89 --/active    org.cauldron.newton.boot.control
   90 --/resolved  chainlink-api-bundle    91 --/active    Spring Chain
Cli
   92 --/resolved  slf4j-api               93 --/resolved  slf4j-log4j12
   94 --/resolved  aopalliance.osgi        95 --/resolved
jcl104-over-slf4j
   96 --/resolved  spring-core             97 --/resolved
```

```
spring-osgi-io
   98 --/resolved  spring-beans           99 --/resolved  spring-aop
  100 --/resolved  spring-context        101 --/resolved
spring-osgi-core
```

Note that as well as installing springchaincli (bundle 91 in this list), the Spring-OSGi runtime has also been installed. Newton does this on demand whenever a Spring-OSGi bundle is installed. To see the OSGi services published by these bundles type services -i, or services -l <bundle-number> for more detail

Rerunning *help*

```
>help
Available command groups (type 'enter' to enter a group):
system - commands for manipulating systems
storagecds - Commands for manipulating cds storage
springchain - Spring: Commands for manipulating chains of Links
sleep - sleeps for a configurable number of milliseconds
session - Session commands built into the console
provisioner - Commands for manipulating the provisioner
obr - OBR commands
logman - Commands for controlling logging within container
logconfig - Configuration commands for the log.
log - Log commands
installer - installs/uninstalls/tracks SCA components in this jvm
indexcds - Commands for manipulating the index service
framework - Framework commands
exec - executes a script from a file or url location
configuration - Configuration commands
cds - Commands for manipulating cds content
binder - Tools for introspecting the binding graph
```

We can see that there is now an entry called springchain. This is the command line extension we'll use to run the demo. If we try it now all we see is

```
> springchain chain
[end of chain]
```

This is because there are no links installed. We can get one by installing springlink

```
>installer install composite:springlink
```

Re-running the demo we now see

```
> springchain chain
chain: cli->{Local-A-Default} [end of chain]
```

This indicates that we have a chain consisting of one springlink. The link has bound to the command line because it has property tag=a which is accepted by the filter (tag=a) on springchaincli's Link reference.

SCA composite documents can be used to override or extends the SCA template files embedded in Newton bundles. In the chain above the name 'Local-A-Default' comes from the property 'linkName' set in the SCA template. We're going to override this using an SCA composite called local-link-a.composite.

```
<composite name="local-link-a">
<bundle.root bundle="org.cauldron.newton.examples.spring.chain.link"
  version="1.0" />
  <include name="springlink" />
```

```
  <component name="springLink">
    <property name="linkName" value="LocalLink-A" type="string" />
  </component>
</composite>
```

The bundle.root element points at the bundle this composite will be created by. The include element selects the template being overridden. The component element identifies component being overridden and includes a new value for the overridden property.

Only one instance of any bundle can be installed per OSGi framework instance, so before we can install local-link-a.composite we have to uninstall the default springlink composite.

```
> installer uninstall composite:springlink
```

We can now install the new composite using

```
> installer install examples/spring/xml/local-link-a.composite
```

Re-running springchain we can see overridden name ' LocalLink-A' has been picked up.

```
> springchain chain
chain: cli->{LocalLink-A} [end of chain]
```

Notice in the above that the default composite for a Spring-OSGi bundle, or any Newton bundle, is installed using a logical name of the form composite:<template-name>. On the other hand, overriding composites, which live outside the bundle they configure, are installed by providing the path or URL of their composite file.

Next we'll uninstall everything.

```
> installer uninstall examples/spring/xml/local-link-a.composite
> installer uninstall composite:springchaincli
```

Having done this Newton will mark all unused bundles as available for garbage collection, and, in due course, uninstall them. Further, Newton notices that the Spring-OSGi runtime is no longer needed, so its bundles are also stopped and made eligible for garbage collection.

Take a look at the installed bundles now using bundles -i. Within 30 seconds the list should have returned to that seen at the start of the demo, so that we have a blank Newton instance once again. You can run

```
> installer gc
```

to speed up the garbage collection process, although it still wont occur instantly.

3.4.5. Remote Example

The previous example was confined to a single Newton instance, and we did everything manually. Next we're going to build a chain spanning multiple Newton instances. To do this we'll use Newton to promote OSGi level services to SCA level, and then show how an SCA system description can be used to manage the distributed chain.

Before doing anything run the ant task that replaces the local chain metadata with distributed chain metadata.

```
> ant use.distributed.chain
```

This copies the distributed chain metadata into the bundle resources directory (src/main/resources in each bundle module). If you are using an IDE, you should refresh your projects after running this task.

Next rebuild the examples

```
> ant clean.examples install.examples
```

Next let's look at how the metadata has changed. In the spring specific files, below META_INF/spring the only things that have changed are the OSGi filters and properties on the references and services of type Link. The filters now have the form

```
filter="(newton.sca.reference=nextLink)"
```

and the properties now have the form

```
<service-properties>
  <entry key="newton.sca.service" value="springLink"/>
</service-properties>
```

The filter name newton.sca.reference and property name newton.sca.service are handled in a special way by Newton.

Services with the newton.sca.service property are published at SCA level rather than OSGi level. The value of the property is used as the SCA service name.

References with the newton.sca.reference filter are resolved at SCA level rather than OSGi level. The value of the filter is used as the SCA reference name.

Note:

From their point of view, Spring-OSGi bundles only ever interact directly with the OSGi registry, so they are oblivious to Newton.

The Newton template files have also changed.

springchaincli.template (namespaces omitted)

```
<composite name="springchaincli" >
  <component name="cli">
    <reference name="nextLink" />
    <description>spring chain cli</description>
    <implementation.spring />
  </component>

  <reference name="nextLink" multiplicity="0..1">
    <interface.java
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link"
/>
  </reference>

  <wire>
    <source.uri>cli/nextLink</source.uri>
    <target.uri>nextLink</target.uri>
  </wire>
</composite>
```

The first thing to notice is that the springchaincli component now contains a reference called nextLink. This was made available by the (newton.sca.reference=nextLink) filter which promoted nextLink to SCA level. The wire element connects this component level reference to a composite level reference of the same type. At the moment no SCA binding is specified for the composite level reference. This will be supplied later by composite instance files.

springlink.template (namespaces omitted)

```
<composite name="springlink">
  <reference name="nextLink" multiplicity="0..1">
    <interface.java
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link"
/>
  </reference>
  <service name="link">
    <interface.java
interface="org.cauldron.newton.test.bundles.chainlink.interfaces.Link"
/>
  </service>
  <component name="springLink">
    <reference name="nextLink" />
    <service name="link" />
    <description>spring based chain link</description>
    <property name="service.pid"
              value="org.cauldron.newton.examples.spring.chain.link"
              type="string" />
    <implementation.spring />
  </component>

  <wire>
    <source.uri>springLink/nextLink</source.uri>
    <target.uri>nextLink</target.uri>
  </wire>>

  <wire>
    <source.uri>link</source.uri>
    <target.uri>springLink/link</target.uri>
  </wire>
</composite>
```

The springLink component includes a reference called nextLink and a service called springLink, both promoted from the Spring-OSGi bundle. They are wired respectively to a composite level reference and service, each with an unspecified binding. These will be filled in by the composite instance documents.

That's the end of the differences in the metadata embedded in the example bundles. It's also worth looking at some of the composite instance documents we'll be using.

For springchaincli there is only one composite instance document, here it is

springchaincli.composite (namespaces omitted)

```
<composite name="springchaincli" >

  <bundle.root bundle="org.cauldron.newton.examples.spring.chain.cli"
```

```
                version="1.0" />
  <include name="springchaincli" />
  <reference name="nextLink" >
    <binding.rmi filter="(tag=a)" />
  </reference>
</composite>
```

As in the local example this overrides some of the template. In this case it adds an RMI binding to the nextLink reference, i.e. tries to resolve the reference using an RMI based transport, and gives it a filter of (tag=a). This filter will make springlinkcli connect to the first link in the chain (remote-link-a).

For springlink there are three composite instance documents, all of which have the same form. The first is shown below

remote-link-a.composite (namespaces omitted)

```
<composite name="remote-link-a">
  <bundle.root bundle="org.cauldron.newton.examples.spring.chain.link"
                version="1.0" />
  <include name="springlink" />

  <reference name="nextLink" >
    <binding.rmi filter="(tag=b)" />
  </reference>

  <service name="link">
    <binding.rmi >
      <attribute name="tag" value="a" type="string" />
    </binding.rmi>
  </service>
  <component name="springLink">
    <property name="linkName" value="Remote-Link-A" type="string" />
  </component>
</composite>
```

In this case several template level settings have been overridden. The bindings are all set to RMI. The service property tag=a is added, and the reference filter is set to (tag=b), so the composite provides link-a and tries to connect to link-b, all over RMI. Also, the linkName property is set to Remote-Link-A.

remote-link-b.composite and remote-link-c.composite differ only on their filters, properties and linkName property value. These are arranged so that the chain wires up in alphabetical order.

3.4.5.1. Running the remote example

To run the example we'll need three running Newton instances. The first of these should be on your development machine. The others can be on the same machine, or on different machines, as long as all machines are within multicast range of each other.

Assuming that they are all being started on one machine from the same Newton install directory the commands we need are as follows. You should pick a fabricName unique to yourself to prevent remote interaction with other Newton instances

for unix

development machine

```
$ ./container -fabricName=your-fabric-name
```

instance 1

```
$ ./container -instance=1 -fabricName=your-fabric-name
```

instance 2

```
$ ./container -instance=2 -fabricName=your-fabric-name
```

for windows:

development machine

```
> container.bat -fabricName=your-fabric-name
```

instance 1

```
> container.bat -instance=1 -fabricName=your-fabric-name
```

instance 2

```
> container.bat -instance=2 -fabricName=your-fabric-name
```

The logs for the different Newton instances can be found below the Newton install at var/container.log, var/1/container.log and var/2/container.log.

At startup, Newton is in local mode. To set it up for distributed use, we must first initialise the distributed runtime and install its command line extensions by running the following on the development machine (only).

```
> exec etc/scripts/single-jvm-dist-infra
```

A side effect of running this script is to start Newton's global event log. This is on your development machine below the Newton install directory at var/global-event.log. Warnings and errors from any Newton instance in you fabric can be found here.

We should also load the example bundles into CDS.

```
> cds scan remote examples/spring/lib
> cds scan remote examples/chainlink/build/lib
> cds publish remote examples/spring/xml/publish-springdm-1.1.2.xml
<Spring-OSGi install directory>
```

Note that this time we're loading bundles into the remote zone to make them available to all Newton instances. Make sure you enter remote here, not boot.

The other two Newton instances can share the distributed runtime installed in the first one, but have to be told to make themselves available for remotely initiated installations. To do this run

```
> system manage etc/systems/remote-container.system
```

in both of them.

First time round we'll install the chain manually as follows:.

On the development machine we'll install springchaincli and an instance of sprinklink

configured to serve as link a.

```
> installer install examples/spring/xml/springchaincli.composite
> installer install examples/spring/xml/remote-link-a.composite
```

In instance 1 we'll install link-b.

```
> installer install examples/spring/xml/remote-link-b.composite
```

In instance 2 we'll install link-c

```
> installer install examples/spring/xml/remote-link-c.composite
```

Returning to the development machine we can now try out the chain

```
> springchain chain
chain: cli->{Remote-Link-A}->{Remote-Link-B}->{Remote-Link-C} [end of
chain]
```

So, without any code changes we've distributed out Spring-OSGi based chain over three Newton instances, and arranged for them to wire up in the correct order.

Before the next part of the demo we need to uninstall all of the springlinks, although we'll keep the springchaincli.

Development machine

```
> installer uninstall examples/spring/xml/remote-link-a.composite
```

Instance 1

```
> installer uninstall examples/spring/xml/remote-link-b.composite
```

Instance 2

```
> installer uninstall examples/spring/xml/remote-link-c.composite
```

The ability to wire up Spring-OSGi services and references remotely is valuable, but it's awkward and unscaleable to have to go to each terminal and start the required composites manually. Newton's provisioning system can help here.

Newton is able to deploy and maintain SCA system documents. These documents include references to several related composites and use replication handlers, to control how many of each are created. System documents are handed to Newton's distributed provisioning system. This examines the available containers and based on what it finds tries to satisfy the target state specified by the system document.

We'll demonstrate all of this here by creating a system document to represent the chain of springlinks. Here's the springchain.system system document (links b and c have the same structure as link a, so they've been elided for clarity.

```
<system name="springchain" boundary="fabric">
  <description>spring DM for OSGi based chain</description>

  <system.composite name="linka"
bundle="org.cauldron.newton.examples.spring.chain.link"
                    template="springlink"
                    version="1.0" >

    <reference name="nextLink" >
      <binding.rmi filter="(tag=b)" />
```

```
    </reference>

    <service name="link">
      <binding.rmi >
      <attribute name="tag"
                 value="a"
                 type="string" />
      </binding.rmi>
    </service>

    <component name="springLink">
        <property name="linkName"
                  value="SpringLink-A"
                  type="string" />
    </component>
    <replicator name="scale" />

  </system.composite>

  <system.composite name="linkb" ... />
  <system.composite name="linkc" ... />

  <replication.handler name="scale"
type="org.cauldron.newton.system.replication.ScalableReplicationHandler">
    <property name="scaleFactor" value="0.33333" type="float" />
  </replication.handler>
</system>
```

The value 'fabric' for the boundary attribute on the top level system element tells Newton's provisioning system that it can use all containers that are part of this fabric to try to realise the system.

This system includes three composites and one replication handler.

The composites have essentially the same structure as the remote-link-a.composite etc. that we used above. The only differences are that the bundle information and template now appear as attributes, that the link names have the form SpringLink-A etc. and that each composite includes a reference to the replication handler.

The replication handler is a ScalableReplicationHandler. Handlers of this type produce a number of copies that varies with the number of available Newton instances according to the specified scale factor. In this case the scale factor is 0.33333, because we want a third as many copies of each link type as there are containers.

Before asking Newton to manage this system, we'll take a look at the state of the provisioner.

```
> remote-provisioner status
-----------------
Provisioner history:
  Resolve(Success)   current: 0 max: 0 total: 0
  Install(Success)   current: 0 max: 0 total: 0
  Uninstall(Success) current: 0 max: 0 total: 0
  Resolve(Failure)   current: 0 max: 0 total: 0
  Install(Failure)   current: 0 max: 0 total: 0
  Container(Failure) current: 0 max: 0 total: 0
```

```
  Uninstall(Failure) current: 0 max: 0 total: 0
-----------------
Listing all containers
Container ContainerID[cd9157e0-30ca-12a1-94ca-bbe1b0695873:opensuse-1]
Container ContainerID[75bb8580-30c6-12a1-926b-bbe1b0695873:opensuse-1]
Container ContainerID[0e1895a0-312d-12a1-ad32-07a4b50f4af5:opensuse-1:2]
Container ContainerID[007d00c0-312d-12a1-9efb-595061512df3:opensuse-1:1]
-----------------
No commands registered
-----------------
```

Here you should see four containers, one for each Newton instance you started, and one used internally by Newton. If you don't see all four make sure you have three Newton instances running and that you have run

```
system manage etc/systems/remote-container.system
```

on instances 1 and 2.

As you can see from the last line, the provisioner has not yet been given any commands.

Now let's tell Newton to manage our system

```
remote-system manage examples/spring/xml/springchain.system
```

Taking another look at the provisioner state you should see the following.

remote-provisioner status

```
> remote-provisioner status
-----------------
Provisioner history:
  Resolve(Success)   current: 3 max: 3 total: 3
  Install(Success)   current: 3 max: 3 total: 3
  Uninstall(Success) current: 0 max: 0 total: 0
  Resolve(Failure)   current: 0 max: 0 total: 0
  Install(Failure)   current: 0 max: 0 total: 0
  Container(Failure) current: 0 max: 0 total: 0
  Uninstall(Failure) current: 0 max: 0 total: 0
-----------------
Listing all containers
Container ContainerID[40204160-3132-12a1-9f58-595061512df3:opensuse-1:1]
Container ContainerID[cd9157e0-30ca-12a1-94ca-bbe1b0695873:opensuse-1]
Container ContainerID[75bb8580-30c6-12a1-926b-bbe1b0695873:opensuse-1]
Container ContainerID[0e1895a0-312d-12a1-ad32-07a4b50f4af5:opensuse-1:2]
-----------------
Listing all commands
springchain:linka : Hosting -> ContainerID[0e1895a0-312d-12a1-ad32-
07a4b50f4af5:opensuse-1:2] cost = 1
springchain:linkb : Hosting ->
ContainerID[cd9157e0-30ca-12a1-94ca-bbe1b0695873:opensuse-1] cost = 1
springchain:linkc : Hosting ->
ContainerID[40204160-3132-12a1-9f58-595061512df3:opensuse-1:1] cost = 1
-----------------
```

If you ran this command very quickly, you may see different status messages, but it will settle down to something like this after a few seconds.

Here we can see that the provisioner has been asked to install one copy each of links a,b and c, and that it has succeeded in doing so.

To verify this let's run the chain again

```
> springchain chain
chain: cli->{SpringLink-A}->{SpringLink-B}->{SpringLink-C} [end of
chain]
```

If you go to your Newton instance-1 or instance-2 now and type bundles -i or services -i, you'll be able to see that the Spring-OSGi runtime and the springlink bundle have been installed.

When we're finished with the system we can shut it down by retiring it from the provisioner as follows.

```
remote-system retire examples/spring/xml/springchain.system
```

If you now check the provisioner status using **remote-provisioner status** you should see that it is back to its initial state, with no commands registered. To verify that the chain has really gone away.

let's run it again

```
> springchain chain
[end of chain]
```

What we've done here is describe a full distributed system based on Spring-OSGi bundles using a single SCA system document, used Newton to automate the details of deploying it across several OSGi containers, tested it and then used Newton to remove it again.

For the purposes of this example we started Newton's distributed runtime on a single machine. In production deployments of Newton this runtime is typically deployed in a replicating and self healing configuration.

3.4.6. Summary

The important points to take away from these examples are that

- Using Spring-OSGi bundles with Newton involves no changes to code and lightweight changes to configuration
- Newton provides an easy way to configure and deploy Spring-OSGi bundles
- Newton is capable of managing the deployment and wire-up of multiple Spring-OSGi bundles to realize the application described in a high level SCA system description.

Although we didn't show it here, services and references based on Spring-OSGi bundles can interact with SCA level services from any SCA composite deployable in Newton, not only those built using Spring-OSGi.

3.5. Scatter Gather Demo

3.5.1. Summary

This demo uses Newton to build a traditional scatter-gather compute grid. The grid uses a

number of distributed worker composites coordinating via a Java Space.

The aim of this demo is to build such a grid manually, explicitly installing the required composites in three separate Newton containers. In the Scatter Gather System Demo the same grid is set up using Newton Environments which automate much of what we do here and improve resilience.

This is not supposed to be a real world grid, in particular it doesn't make use of transactions or persistence to ensure delivery of all submitted tasks.

3.5.2. Overview

In this demo there are 3 composite types.

1. A Javaspace composite that acts as a coordination point for the grid
2. A Client that submits tasks to the grid
3. Several Worker composites that process the tasks and return the result
4. A Command Line Interface (CLI) that can be used to trigger the submission of work by the client.

The way the the demo works is that the client breaks the work it wants done down into tasks and writes entries wrapping these into the Javaspace. Meanwhile idle Workers are monitoring the space for new tasks. Whenever they spot one they take it, process it and return an entry wrapping the result to the space. Finally the client monitors the space for entries wrapping task results until it has taken one for every task it submitted, it then collates these and prints out the result.

Each of the composites listed above is deployed in its own factory bundle. The client and worker import a common API package from an API bundle. This API bundle and the client have an RMI-Codebase manifest header identifying the code needed to deserialize the objects sent over the network. To see the detail consult the component.bundle tasks of the ant build.xml file of the demo source code.

3.5.3. Running the demo.

3.5.3.1. Prerequisites

Try the local Chainlink Demo first - this walks through some basic Newton concepts that are not covered here.

3.5.3.2. Booting the Newton container

In order to run the demo it is first necessary to start three Newton containers in the same fabric. Fabrics are used to give scope to distributed containers. Only containers belonging to the same fabric can see each others services. To start three containers run:

In terminal 1

```
$ ./container -fabricName=myfabric -instance=1
```

In terminal 2

```
$ ./container -fabricName=myfabric -instance=2
```

In terminal 3

```
$ ./container -fabricName=myfabric -instance=3
```

It is important that you remember to use your own fabric name, not "myfabric".

The -instance flag is used to separate the state of the different Newton containers. This makes it straightforward to run this demo on a single machine. You can also run the different Newton instances on different machines as long as your firewall is configured to forward the multicast packets Newton uses for discovery.

After starting Newton wait for the "Boot complete" message, after which Newton will make a command line available.

3.5.3.3. Booting the distributed runtime

Newton's distributed infrastructure makes use of multicast, so it is essential that your network and computers are not configured to block multicast traffic. If you are unsure of your setup, or having difficulty running this example the please see Troubleshooting Multicast.

Every instance of Newton starts up with the client side of the distributed infrastructure already installed. In this example we will manually boot the server side infrastructure. In *one* of the Newton containers run

```
> installer install etc/instances/reggie.composite
> installer install etc/instances/server-cds.composite
```

This installs the remote services registry, which Newton uses to detect and wire up services in different JVMs. It also starts the *remote zone* of CDS. All Newton containers share a common view of the content in CDS's remote zone.

Note:

This remote infrastructure is itself implemented as a set of Newton composites.

Optionally you can start the Jini browser composite, a GUI tool for viewing the remote registry. You can use this to see the services that available to your group remotely.

```
> installer install etc/instances/jinibrowser.composite
```

The services may take a few seconds to come up - you'll get an error message at the next step if you are too quick.

3.5.3.4. Loading bundles into CDS

At this point the bundles used by the scatter-gather demo are not present in the CDS, so they must be loaded. Note that this doesn't install anything, it just makes resources available for future composite installations.

To load the demo bundles run the following in one of your Newton instances.

```
> cds scan remote examples/scattergather/build/lib
```

This loads all of the bundles in demo/build/lib into the *remote zone* of CDS, extracting metadata and indexing the bundles as they are loaded. Resources in the remote zone can be seen by all Newton containers in the same fabric.

As well as the demo bundles we are going to be using a Javaspace. There are currently two Javaspaces wrapped as Newton composite, Outrigger, the Jini TSK reference implementation, and Blitz, the leading open source implementation. These ship with the main Newton install, not with the demos, so their bundles must be loaded into CDS separately as follows:

```
> cds publish remote etc/cds/jini-service-bundles.xml
```

Here a CDS Publish File has been used to load the bundles into the remote Zone of CDS.

3.5.3.5. Running the demo

The rest of this demo is split over the three Newton containers that are now running.

In the container 1 install the Client and CLI composites as follows.

```
> installer install examples/scattergather/build/etc/cli.composite
> installer install examples/scattergather/build/etc/client.composite
```

In container 2 install the javaspace. Choose *one* of the following. *Either* Outrigger:

```
> installer install etc/instances/outrigger.composite
```

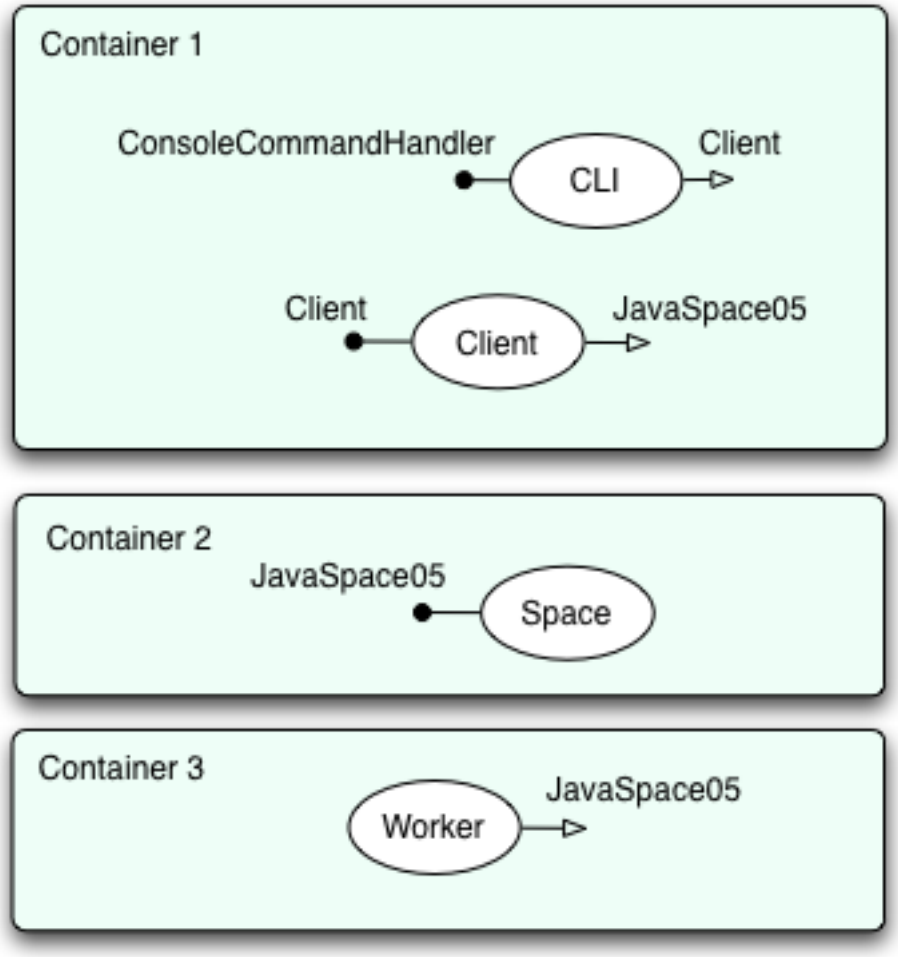
or Blitz:

```
> installer install etc/instances/blitz.composite
```

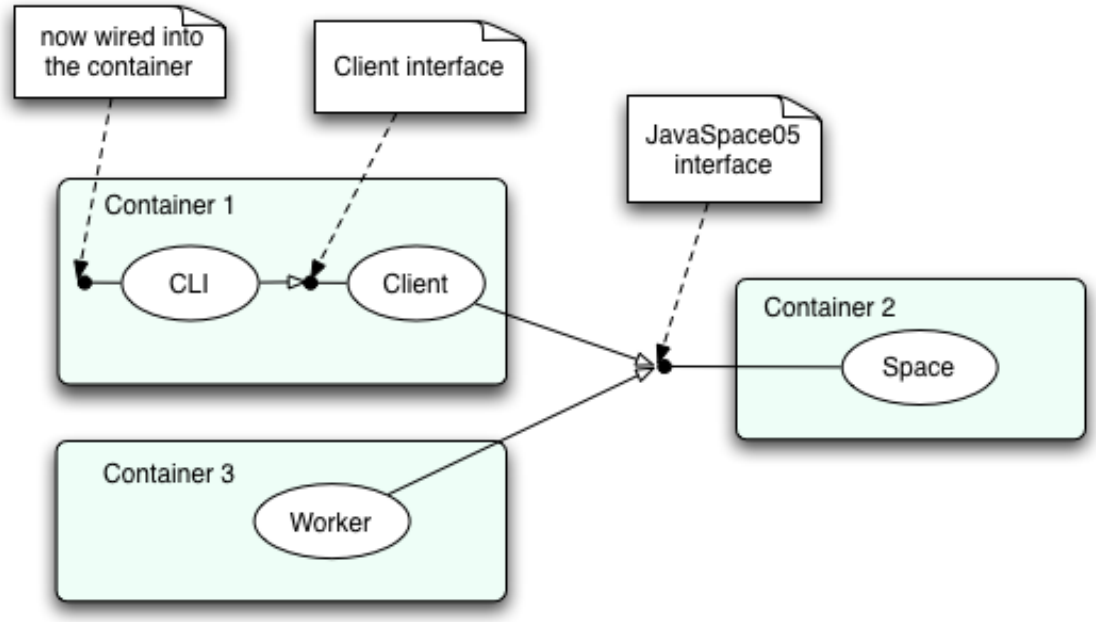
In the third container install a worker as follows:

```
> installer install examples/scattergather/build/etc/worker.composite
```

At this point we have set up a scatter-gather grid with one worker. The structure of the composites we've installed an their distribution among the containers is shown below.



As the composites are created Newton examines the their services and references and uses them to dynamically wire up the composites in the following arrangement.



Note:

The CLI is now wired into the container that created it and has added a new command group called `scattergather` to the Newton command line. To see this just type `help` in container 1, i.e. the container we added the scattergather CLI to.

To see the grid working we can use this CLI to submit some work to the grid.

```
> scattergather submit
[stdout] Submitted 100 tasks
> [stdout] completed 100 tasks in 18218ms
[stdout] status: success
```

The 100 tasks that are submitted don't really do any work, they just sleep for 0.1 seconds, so with one worker the time taken should be 10 seconds. From the output we can see that it actually took more than 18 seconds. Most of this is due to first time initalisation of the Javaspace, although there will always be some infrastructure overhead. Running it again gets us nearer to the expected time.

```
> scattergather submit
[stdout] Submitted 100 tasks
> [stdout] completed 100 tasks in 11721ms
[stdout] status: success
```

Of course a real grid has more than one worker, so we now install 2 more. We'll do this in container 3, although they could be anywhere.

```
> installer install examples/scattergather/build/etc/worker.composite
> installer install examples/scattergather/build/etc/worker.composite
```

Now we resubmit the 100 tasks to see the benefit of the extra workers.

```
> scattergather submit
[stdout] Submitted 100 tasks
```

```
> [stdout] completed 100 tasks in 4947ms
[stdout] status: success
```

This time the tasks are processed far faster, taking just under 5 seconds.

3.5.4. Further exercises

1. Try switching Javaspace, e.g. replacing Outrigger with Blitz without stopping the other composites. After the switch submit another job to check everything is still working.
2. Add and Remove more workers to the existing and additional Newon containers, and observe the results.
3. Uninstall everything using `installer uninstall <descrtiptor>`, and use `bundles -i` to confirm that the bundles used by the scatter-gather composites have been garbage collected.

3.6. Scatter Gather System Demo

3.6.1. Overview

The Scatter Gather Demo showed how to start up the scatter gather components by hand to provide a number of worker components to which a client submits a number of jobs via a java space. This is useful up to a point but does not address issues of scaling or reliability. To start to build these properties into the architecture we will make use of the concept of a system of components.

This demo makes use of system managers, provisioners and containers, for more information on the relationships between these services see System Managers, Provisioners and Containers.

3.6.2. Discussion

The following is the xml markup to define a scatter gather worker system:

```
<?xml version="1.0"?>
<system name="scattergather-workers" boundary="fabric">

  <description>An system that sets up one worker for every container in
the grid</description>

  <system.composite name="worker" bundle="sg-worker-bundle"
template="workerTemplate" version="0.1">
    <replicator name="scale" />
  </system.composite>

  <system.composite name="space" bundle="blitz-bundle"
template="blitzTemplate" version="0.1" />

  <replication.handler name="scale"
type="org.cauldron.newton.system.replication.ScalableReplicationHandler">
    <config name="scaleFactor" value="1" type="float"/>
  </replication.handler>

</system>
```

This defines an system that contains one Blitz JavaSpace and a scalable set of workers. In this case the workers will scale to match the number of available containers in the network. Note the boundary="fabric" attribute, see Boundary for more information on system boundaries.

In order to test this system we will use five containers in separate JVM's (possibly on different machines - it is up to you, though bear in mind Multicast visibility). Below are listed the containers we will use and the components that will be installed on them.

- Container 1: Boot Container
  - Reggie
  - CDS Server

    - Remote System CLI
    - Remote Provisioner CLI

- Container 2: Fabric System Manager
    - Remote System Manager
    - Remote Container Registry

- Container 3: Fabric Provisioner
    - Remote Provisioner
    - Remote Container Registry

- Container 4: Fabric Container
    - Remote Container

- Container 5: Fabric Container
    - Remote Container

If you're running on the same machine, you should start each container with a different -instance=N argument. i.e. bin/container -instance=1, bin/container -instance=2, etc.

3.6.3. Running the demo

So let's get started:

3.6.3.1. Container 1

```
$ bin/container -instance=1
...
> installer install etc/instances/reggie.composite
...
> installer install etc/instances/server-cds.composite
...
> installer install etc/instances/remote-manager-cli.composite
...
> installer install etc/instances/remote-provisioner-cli.composite
...
> cds publish remote etc/cds/jini-service-bundles.xml
> cds scan remote examples/scattergather/build/lib
```

This sets up reggie, the server cds and cli components in container1 and scans in the jini service and demo component bundles into the remote cds repository.

3.6.3.2. Container 2

```
$ bin/container -instance=2
...
> installer install etc/instances/remote-registry.composite
> system manage etc/systems/remote-manager.system.xml
```

This sets up a remote registry and the components that make up the system manager.

3.6.3.3. Container 3

```
$ bin/container -instance=3
...
> installer install etc/instances/remote-registry.composite
> system manage etc/systems/remote-provisioner.system.xml
```

This sets up a remote registry and a remote provisoner components.

Now to submit the scatter gather worker system.

3.6.3.4. Container 1

```
> remote-system manage examples/scattergather/build/etc/worker.system
```

In this case you have submitted the system but there are no containers on which to install the components, so the system waits. Lets start up a container.

3.6.3.5. Container 4

```
$ bin/container -instance=4
...
> system manage etc/systems/remote-container.system.xml
```

The system will notice the new remote container and install the blitz instance and one worker to it. You can check this by using the remote provisioner cli:

3.6.3.6. Container 1

```
> remote-provisioner status
```

Now lets add a second container:

3.6.3.7. Container 5

```
$ bin/container -instance=5
...
> system manage etc/systems/remote-container.system.xml
```

Again the system will notice and install a second worker instance to the container. You can now submit jobs via the client to both of these workers.

3.6.3.8. Container 1

```
> installer install examples/scattergather/build/etc/client.composite
> installer install examples/scattergather/build/etc/cli.composite
> scattergather submit
```

Now to test resilience. You can stop one of the containers and the system components will be dynamically reprovisioned to the other container. Lets stop the first container that is hosting blitz:

3.6.3.9. Container 4

```
> system retire etc/systems/remote-container.system.xml
```

You should see blitz startup on container 5. If you restart container 4 you should see a

second worker installed.

When you are finished with the scatter gather components you can remove them from the containers using the following commands:

3.6.3.10. Container 1

```
> installer uninstall examples/scattergather/build/etc/cli.composite
> installer uninstall examples/scattergather/build/etc/client.composite
> remote-system retire examples/scattergather/build/etc/worker.system
```

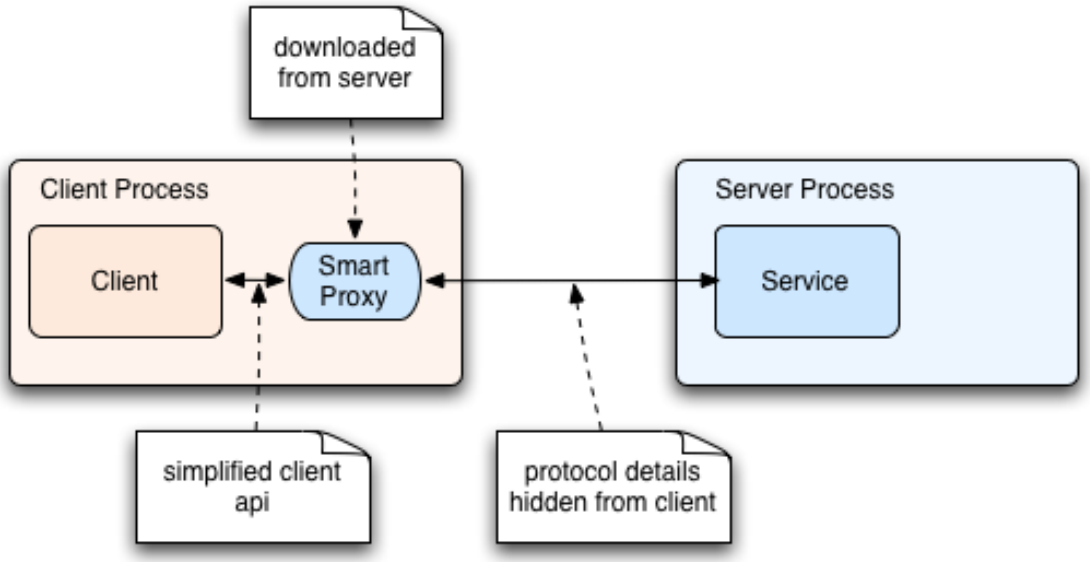
3.7. Smart Proxy Demo

3.7.1. Summary

This example shows how Newton can be used to wire up Jini based smart proxies.

Smart proxies are written by service providers and transparently downloaded by clients when they start to use the service. They allow service providers to take responsibility for both sides of the remote protocol their service implements, and to present a simplified facade to their clients.

The basic model is shown below.



Note:

In this context the emphasis of the term 'remote protocol' isn't so much on the transport level protocol (although it certainly includes this), as on the low level code dealing with error and timeout handling, caching, batching and other remoting details that are best hidden from the client. Without smart proxies each client would have to either manually install a custom client library for each service it used, or handle all of the protocol details itself. Either way the client would be hard wired to a particular set of protocol details and so much less able to live in a dynamic services world full of alternate service implementations and regular service upgrades.

These issues were well understood by Jini's creators, and smart proxies have always been a central Jini theme.

Newton tries to simplify the process of writing and deploying smart proxies so as to make them accessible to developers with no understanding of Jini. A little RMI knowledge is required, but not very much.

3.7.2. Overview

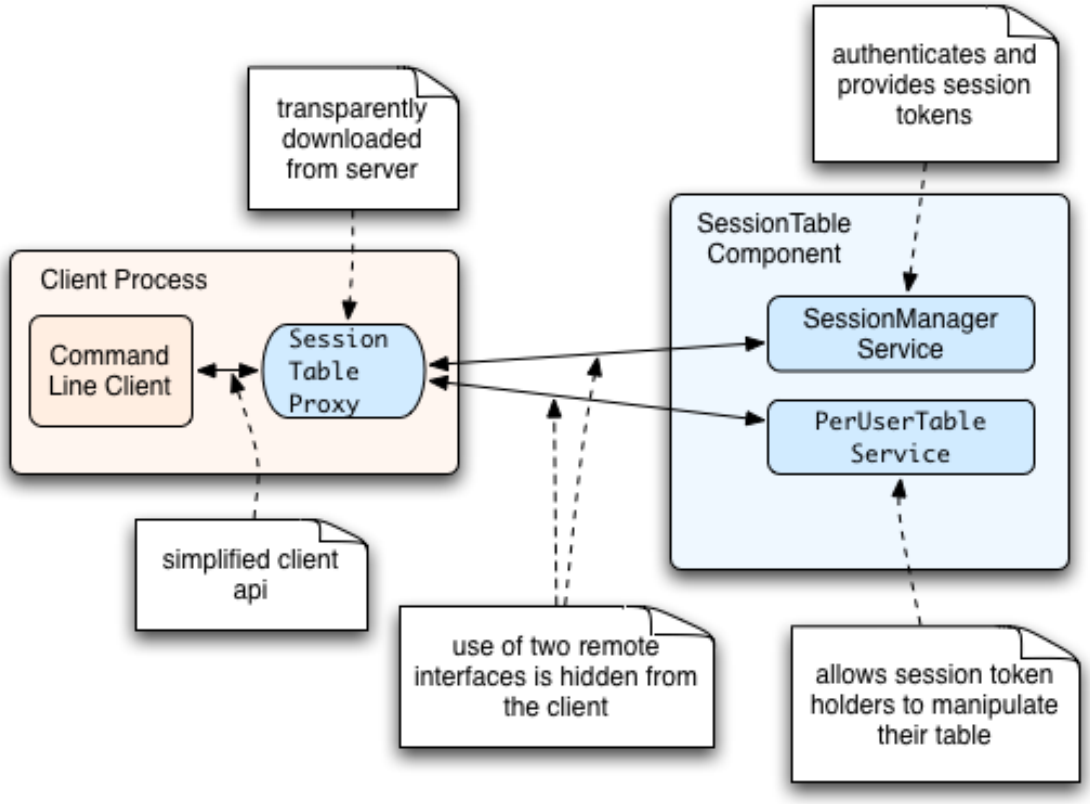
In this demo the `Server` composite constains two local components:

- a `SessionManager` component that provides authenticated users with `SessionToken` credential objects.
- a `PerUseTable` component that allows holders of an authenticated `SessionToken` to manipulate their private name-value table

It also provides a smart proxy component called `SessionTable`. This has a single interface that combines both login and table manipulation, simplifying things for the client by hiding the fact that there are two remote interfaces and the details of `SessionToken` handling.

A CLI based client is used to communicate with this service. It binds to a SessionTable reference, unaware that it is implemented as a smart proxy.

The system we are going to create is shown below.



The CLI descriptor and code is much as in the other examples so we'll only look at the `Server` composite in detail.

The `Server` composite creates its `SessionManager` and `PerUseTable` components in the usual way. The `SessionTable` smart proxy component is created in the same way, but uses a special `jini-proxy` implementation type. It It provides a `SessionTable` service at Jini scope in the usual way. The full `server` template XML is shown below.

```
<?xml version="1.0"?>
<composite name="serverTemplate">
    <description>session table service template</description>

    <service name="sessionTable-export">
        <interface.java
interface="org.cauldron.newton.example.sessiontable.proxyapi.SessionTable"/>
        <binding.rmi/>
    </service>

    <component name="sessionManager">
        <description>session manager</description>
        <implementation.java.callback
impl="org.cauldron.newton.example.sessiontable.server.SessionManagerImpl"/>
    </component>

    <component name="perUserTable">
        <description>session scoped table</description>
        <interface.java name="perUserTable"
interface="org.cauldron.newton.example.sessiontable.serverapi.PerUserTable"/>
        <reference name="sessionManager"/>
        <implementation.java.callback
impl="org.cauldron.newton.example.sessiontable.server.PerUserTableImpl"/>
    </component>

    <!-- remote proxy -->
        <component name="sessionTable">
        <description>session scoped table proxy</description>
        <reference name="sessionManager"/>
        <reference name="perUserTable"/>
        <implementation.jini.proxy
impl="org.cauldron.newton.example.sessiontable.proxy.SessionTableProxy"/>
    </component>

    <wire>
        <source.uri>perUserTable/sessionManager</source.uri>
        <target.uri>sessionManager</target.uri>
    </wire>

    <wire>
        <source.uri>sessionTable/perUserTable</source.uri>
        <target.uri>perUserTable</target.uri>
    </wire>

    <wire>
        <source.uri>sessionTable/sessionManager</source.uri>
        <target.uri>sessionManager</target.uri>
    </wire>

    <wire>
        <source.uri>sessionTable-export</source.uri>
        <target.uri>sessionTable</target.uri>
```

```
    </wire>
</composite>
```

Here the `<implementation.jini.proxy ..>` tag implies the same lifecycle as in the `<implementation.java.callback ..>` case, with the following important differences:

- All of its references must be satisfied by components within its own composite. This is because once a proxy leaves the composite that created it it will no longer be able to receive updates if the composites's external connections change, whereas internal connections are static and thus unaffected by this problem.
- All of its referenced interfaces are automatically remoted at JVM level, and the proxy is passed these as remote stubs rather than as local references. This is so that the proxy can still communicate with its dependencies after it has left the component that created it.
- The proxy should not be referred to by any other component in the composite. This is because once it has left the originating JVM the local copy will not be in sync with deserialized remote copies of the service.

Note that if any of the proxies referenced interaces do not implement `Remote` then they will be transparently wrapped prior to being exported, and throw unchecked `ConnectionException` instances rather than `RemoteException` instances to indicate remoting errors.

As in all of the Jini demos an RMI-Codebase manifest attribute must be set so that the proxy can be deserialized. See the ant build.xml file for this demo for more details.

3.7.3. Running the example.

3.7.3.1. Prerequisites

Try the Local Chainlink Demo and Remote Chainlink Demo before this one. These will introduce you to The Newton Component Model and the basics of Jini in Newton.

3.7.3.2. Booting two Newton containers

In order to run the demo it is first necessary to start two Newton containers. In order to see each other these need to be running as part of the same fabric. To start the two containers on a single machine open two terminals in the Newton base directory and run

In terminal 1

```
$ bin/container -console -fabricName=myfabric -instance=1
```

In terminal 2

```
$ bin/container -console -fabricName=myfabric -instance=2
```

Note that this demo will also work when running across several machines (firewalls

permitting)

Here we've used the -console flag to tell newton to run as a console application rather than as a service.

The -fabricName flag sets the fabric name. We've used 'myfabric' but you should choose your own name to prevent clashes. Note that if the fabric name is not set it defaults to the current user name. This means that simple tests on a single machine will work without the fabricName being set. However it should always be set when more than one machine is involved.

The -instance flag is used to separate the state belonging to the two conatiner instances. This is only necessary because we are running the two instances out of the same Newton install. If the instances were running on different machines there would be no need for this flag.

After starting Newton wait for the "Boot complete" message, after which Newton makes a command line available.

3.7.3.3. Booting the distributed runtime

Newton's distributed infrastructure makes use of multicast, so it is essential that your network and computers are in a configuration that allows multicast traffic. If you are unsure of your setup, or having difficulty running this example the please see Troubleshooting Multicast.

Every instance of Newton starts up with the client side of the distributed infrastructure already installed. At present it is necessary to manually boot the server side infrastructure. In *one* of the Newton containers run

```
> installer install etc/instances/reggie.composite
> installer install etc/instances/server-cds.composite
```

This installs the remote services registry, which Newton uses to detect and wire up services in different JVMs. It also starts the *remote zone* of CDS. All Newton containers share a common view of the content in CDS's remote zone.

This remote infrastructure is itself implemented as an SCA system built from Newton composites.

Optionally you can start the jinibrowser composite, a GUI tool for viewing the remote registry. You can use this to see the services that available to your group remotely.

```
> installer install etc/instances/jinibrowser.composite
```

The services may take a few seconds to come up - you'll get an error message at the next step if you are too quick.

3.7.3.4. Loading bundles into CDS

At this point the bundles used by the session table demo are not present in the CDS, so

they must be loaded. Note that this doesn't install anything, it just makes resources available for future installations.

To load the demo bundles run the following in *one* of your Newton instances.

```
> cds scan remote examples/sessiontable/build/lib
```

This loads all of the bundles in demo/build/lib into the *remote zone* of CDS, extracting metadata and indexing the bundles as they are loaded. Resources in the remote zone can be seen by all Newton containers in the same fabric.

3.7.3.5. Using SessionTable

First create the CLI part on container 1.

```
> installer install examples/sessiontable/build/etc/cli.composite
```

If you now type `help` you'll see a new command group called `sessiontable`. This CLI composite has a reference to a `SessionTable` interface. At the moment there is nothing for it to connect to so lets start the `Server` composite in container 2.

```
> installer install examples/sessiontable/build/etc/server.composite
```

If you opted to run the jinibrowser composite you'll be able to see the exported proxy there.

Newton detects the presence of the `SessionTable` export and wires it up to the CLI. This involves downloading the smart proxy into the CLI's JVM and deserializing it. The CLI composite doesn't have to worry about these details, it just references the interface.

We now have the system shown in the diagram above and can run some tests with it. First, in container 1, enter the sessiontable command group as follwos:

```
> enter sessiontable
```

You can leave the command group at any time by typing `leave`

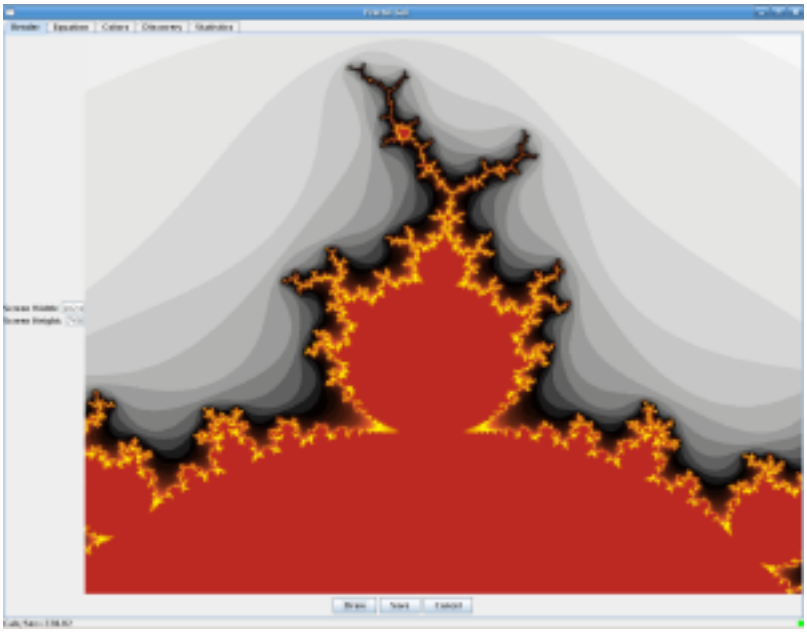
The available commands are `login` to switch user `get` to get values in a user's table and `put` to put values in a user's table. The following session manipulates the private tables of two users.

```
sessiontable> login fred pass
logged in
sessiontable> put age 51
sessiontable> get age
age->51
sessiontable> put name fred
sessiontable> get name
name->fred
sessiontable> login bill pass
logged in
sessiontable> get age
age->null
sessiontable> get name
name->null
sessiontable> put age 34
sessiontable> put name bill
```

```
sessiontable> get age
age->34
sessiontable> get name
name->bill
sessiontable> login fred pass
logged in
sessiontable> get age
age->51
sessiontable> get name
name->fred
```

3.8. Fractal Render Demo

This demo of the Newton framework builds on concepts from previous demos. It shows you how you can build an application which incorporates GUI and networking functionality and also incorporates extensibility patterns. The end goal is to build an application that allows you to compute the values of fractal equations using a self scaling farm of compute servers operating the scatter gather pattern. We will step through various stages to reach this goal, each one demonstrating a feature of the Newton framework.



3.8.1. Environment Configuration

The first thing to note is that though Newton itself is very lightweight in terms of memory usage, this demo is relatively heavyweight due to the queuing nature of the scatter gather worker pattern. Therefore it is advisable to increase the default memory allocation for the Newton JVM. To do this for all Newton JVM's on a single machine you can set the value:

```
JVM_ARGS="-Xmx512m"
```

in "$HOME/.newton/container_env.sh" on unix and "%HOMEDRIVE%%HOMEPATH%\.newton\container_env.bat" on windows.

3.8.2. Initialisation

We will start by constructing a local application, run solely from your development machine. To do this we will firstly startup a Newton container and launch two components; reggie and cds-server.

```
$ cd $NEWTON_HOME
$ bin/container
```

```
...
> installer install etc/instances/reggie.composite
> installer install etc/instances/server-cds.composite
```

Next we load the demo and jini service bundles into CDS. Note we load the content into the remote "zone" of CDS so that later in our demo when we launch other Newton containers they are able to download the demo and jini service bundles from the server-cds component in this container instead of locally adding the bundles to cds in each container.

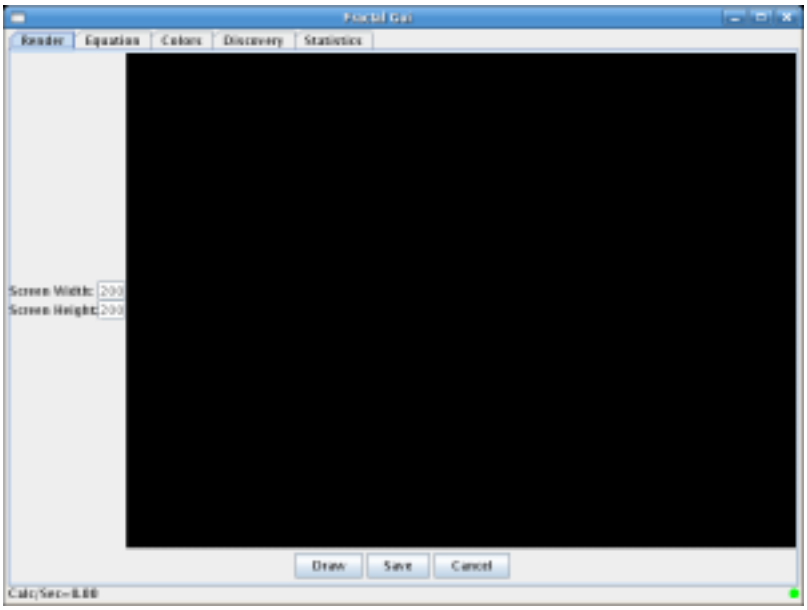
```
> cds scan remote examples/fractal/build/lib
> cds publish remote etc/cds/jini-service-bundles.xml
```

3.8.3. Local rendering

Having initialised the application environment we can then load the first set of components which make up the fractal render demo.

```
> installer install examples/fractal/build/etc/gui.composite
> installer install examples/fractal/build/etc/local-engine.composite
> installer install examples/fractal/build/etc/local-worker.composite
> installer install etc/instances/local-blitz.composite
```

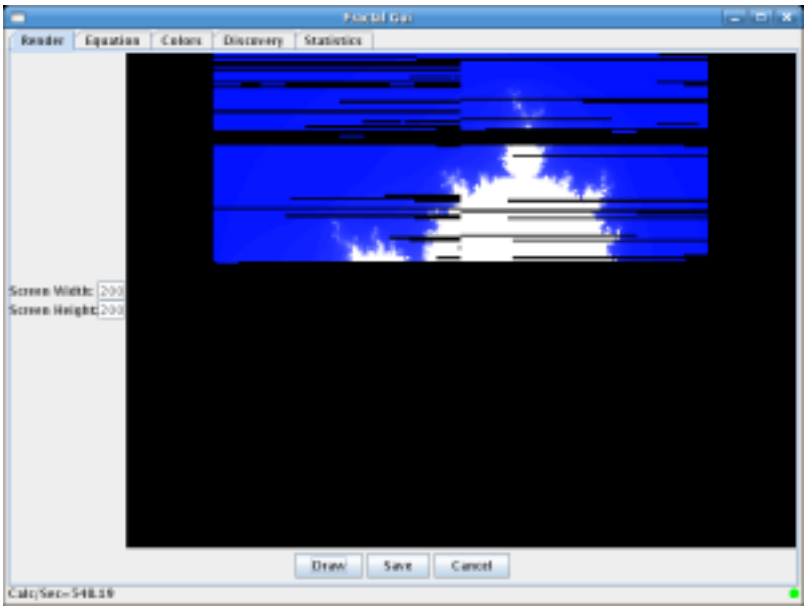
You will then see a window that looks something like the following:



Note:

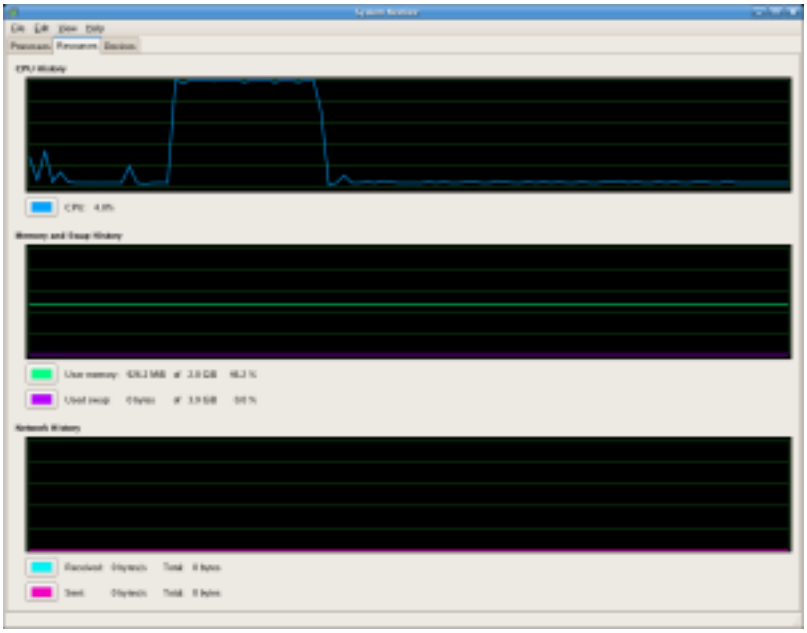
The green icon in the bottom right hand corner of the gui denotes that an engine is connected to the gui and that the engine is connected to a space. If you watch the icon in future steps of the demo you should be able to see it change to amber (when an engine is bound but the engine is not connected to a space) or red when the gui is disconnected from the engine.

Now press the "Draw" button, you should start to see a mandlebrot fractal appear on screen.



If you start a system monitoring tool you should be able to verify that all communications are taking place locally, i.e. no network traffic - all communications are taking place within the Newton container process.

In the figure below the CPU is maxed out but there is no network traffic.



Warning:

In order to capture this screen shot it was necessary to disable the eth0 interface of my machine as there were various background communications non-essential to processing going on. Disabling the eth0 interface also happens to be a very conclusive way of demonstrating there are no remote communications going on.

3.8.4. Remote rendering

We will now move worker and space components onto a separate process, which may or may not be on a separate machine. To do this type the following into your the current Newton container console:

```
> installer uninstall examples/fractal/build/etc/local-engine.composite
> installer uninstall examples/fractal/build/etc/local-worker.composite
> installer uninstall etc/instances/local-blitz.composite
```

This uninstalls the engine, worker and blitz components from this Newton container. You should then install a remote version of the fractal engine in the current Newton container.

```
> installer install examples/fractal/build/etc/engine.composite
```

Here it is worth taking a moment to show the difference between the local and remote engine components. Below are the contents of the files `examples/fractal/build/etc/engine-template.xml`, `examples/fractal/build/etc/local-engine.composite` and `examples/fractal/build/etc/engine.composite`.

3.8.4.1. examples/fractal/build/etc/engine-template.xml

```
<?xml version="1.0"?>
<composite name="engineTemplate">
        <description>Space based engine for fractal demo</description>

        <reference name="space" multiplicity="1..1">
                <interface.java interface="net.jini.space.JavaSpace05"/>
        </reference>

        <service name="engine-export">
                <interface.java
interface="org.cauldron.newton.example.fractal.api.CalculationEngine"/>
                <binding.osgi />
        </service>

        <component name="engine">
                <reference name="space"/>
                <implementation.java.callback
impl="org.cauldron.newton.example.fractal.engine.SpaceCalculationEngine"/>
        </component>

        <wire>
                <source.uri>space</source.uri>
                <target.uri>engine/space</target.uri>
        </wire>

        <wire>
                <source.uri>engine</source.uri>
                <target.uri>engine-export</target.uri>
        </wire>
</composite>
```

You can see that the engine template (upon which each instance is based) imports a reference to a java class that implements the JavaSpace05 interface. However the

template does not define the binding from which to import the reference. This is left up to each instance.

3.8.4.2. examples/fractal/build/etc/local-engine.composite

```
<?xml version="1.0"?>
<composite name="fractal-engine">
        <bundle.root bundle="fractal-engine-bundle" version="1.0"/>
        <include name="engineTemplate"/>

        <reference name="space" >
                <binding.osgi/>
        </reference>
</composite>
```

In the local engine case the reference is `binding.osgi` i.e. bound to the local OSGi registry. Here the Newton framework cause the engine component to connect to an interface published to the osgi registry. If you look inside the `components/jini-starter/resources/templates/blitz-local-template.xml` file you will see that this is exactly what this component does.

3.8.4.3. examples/fractal/build/etc/engine.composite

```
<?xml version="1.0"?>
<composite name="fractal-engine">
        <bundle.root bundle="fractal-engine-bundle" version="1.0"/>
        <include name="engineTemplate"/>

        <reference name="space" >
                <binding.rmi/>
        </reference>
</composite>
```

In the remote engine case the reference binding is `binding.rmi` i.e. services are discovered using the remote service registry and communicate using RMI. Here the engine component connects to a service proxy published to the remote registry with that interface. There are similar differences between the local and remote worker components.

Note:

The only differences between these two cases is in the component definition, no code changes were required. So you can see that in order to change the properties of a binding within Newton (in this case swap from a local to a remote implementation) we only needed to modify xml markup.

This is extremely useful in a number of different situations, for example:

- For developer testing purposes, the developer can test their application purely locally then with a change to their component definition can deploy a tested component across a newtork with no changes to their code
- For administration purposes, the admin can redefine properties of a remote binding.

In future releases of Newton we aim to support other binding mechanisms including WS*, additionally further binding solutions and configuration options are supported by

Paremus's Infiniflow products.

Now we will start one or two other Newton containers. You can do this either on the same machine or a remote machine. However if you are using a remote machine bare in mind each container must be able to resolve multicast traffic between them. See Troubleshooting Multicast if you have problems.

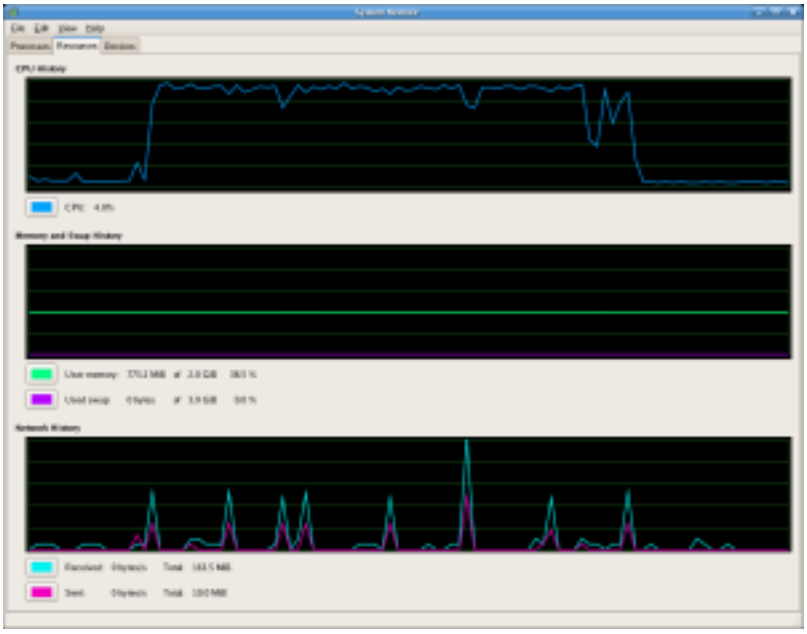
Into these containers we will install remote versions of the worker and space components.

```
$ bin/container
...
> installer install examples/fractal/build/etc/worker.composite
```

and (possibly in a second container):

```
$ bin/container -instance=2
...
> installer install etc/instances/blitz.composite
```

Now when we re draw the mandlebrot equation you should see that the components are communicating over remote interfaces, as shown from the system monitor screenshot below:



3.8.5. Extending the GUI

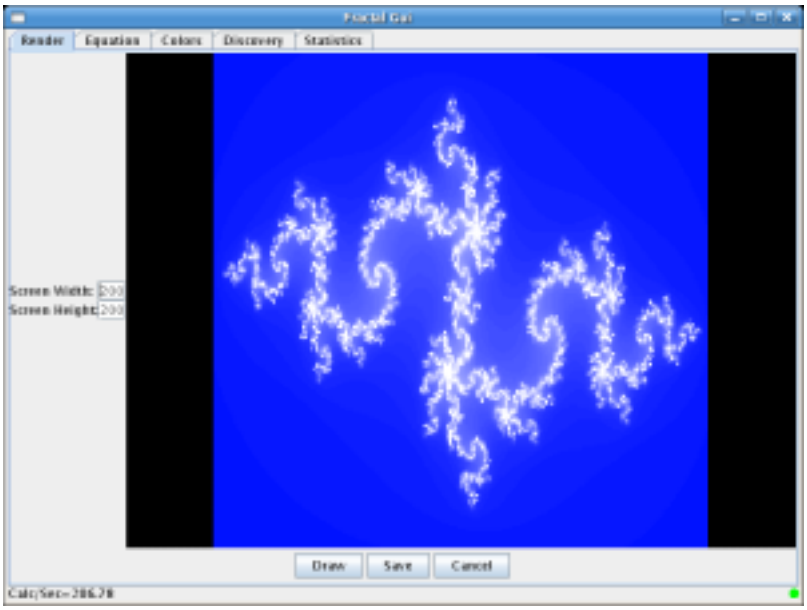
The steps so far show you how to modify the deployment of application components in a network. We will now extend this application with new functionality at runtime, i.e. dynamically add in new functionality to the running application.

If you change tabs to the `Equation` tab at the top of the gui frame and click on the drop down box in the top left hand corner of this tab - you should see that there is only one equation loaded, the mandlebrot equation.

Having verified this we will now add in a Julia equation into our running application. In the console for the Newton container hosting the gui component enter the following:

```
> installer install examples/fractal/build/etc/equation-addons.composite
```

Now if you go to the equations drop-down box you should see that a julia equation has been added, select this, return to the render tab and press draw, you should see the application uses the remote worker to render a Julia fractal image:



3.8.6. Remote rendering extensions

In remote applications it is important to be able to use a transaction manager to ensure that data is not lost when remote components fail. In this demo transactions are optionally supported by the fractal worker component.

3.8.6.1. examples/fractal/build/etc/worker-template.xml

```
<reference name="transaction" multiplicity="0..1" override="no">
    <interface.java
interface="net.jini.core.transaction.server.TransactionManager"/>
    <binding.rmi/>
</reference>
```

This fragment of the worker composite definition we see that the transaction reference is given a multiplicity of 0..1, i.e. the composite can start whether a transaction manager is found or not.

Launch a mahalo component either in one of the existing containers or in a new one:

```
> installer install etc/instances/mahalo.composite
```

It is then possible to demonstrate transactions between the worker and space. If a worker component fails whilst processing data (i.e. the worker component is uninstalled or the

process of the Newton container hosting the worker component is killed). Then assuming the space and mahalo components keep running then the transaction will be rolled back and the image will still render in full despite the loss.

```
> installer uninstall examples/fractal/build/etc/worker.composite
```

3.8.7. Deploying a rendering system

The Newton framework contains a number of tools to manage remote component deployments. One of the most useful is the concept of a System. We will now use these components to build a scalable system of workers across any number of Newton containers. The Newton framework will dynamically detect new containers as they come online and provision workers to them as they are started.

Systems are managed by a collection of components themselves defined via provisioner and system manager systems. These initial systems are deployed on a local container, where as the worker system will be deployed across many (remote) containers.

Firstly pick two containers to host the system manager and provisioner components, these may be new or existing containers. Note the provisioner and system manager may also be installed to the same container - if you do this you only need to install the `remote-registy` component (below) once:

```
> installer install etc/instances/remote-registry.composite
> system manage etc/systems/remote-manager.system
> installer install etc/instances/remote-registry.composite
> system manage etc/systems/remote-provisioner.system
```

These components will discovery remote containers and deploy our worker and space components to them.

Now on each container you wish to add to be added to the worker system you should start a further set of components (defined via a system).

```
> system manage etc/systems/remote-container.system
```

This seems to be swapping one manual install (the worker for a second manual install, the boot-container). However it is possible to make the boot container automatically startup in your container by modifying the boot-scripts for that container. Further information on boot scripts can be found here.

Create a directory `core/boot-scripts/3` and create a file `container.boot` within it. Into this file put the following text

```
# launches a boot-container system that allows remote provisioning of
components
system manage etc/systems/remote-container.system
```

Finally to complete the system manager installation, install a cli to access the boot-system manager into one container.

```
> installer install etc/instances/remote-manager-cli.composite
```

You may now uninstall the worker and space instances you deployed earlier, these will be replaced by a worker system which consists of a single space and one worker per

container.

```
> installer uninstall examples/fractal/build/etc/worker.composite
> installer uninstall etc/instances/blitz.composite
```

In the container you installed the remote-manager-cli component can now type:

```
remote-system manage examples/fractal/build/etc/worker.system
```

Now when you render the image you should be able to see that many containers are taking part in the job of calculating work. You can see this for yourself by looking at the `Discovery` and `Statistics` tabs of the gui.

You can retire all of the worker and space components from the Newton infrastructure using the command:

```
remote-system retire examples/fractal/build/etc/worker.system
```

3.8.8. Extensions

- Try writing your own equations for the fractal gui. Contributions back to the Newton project are appreciated.
- It is also possible to configure Newton to automatically start the provisioner and system manager components. To do this you will need to modify the boot scripts of the Newton container to automatically start these components. This will be described in a follow on demo Newton Bootstrap Demo.

4. Developing

4.1. Newton: Building Composites

This page describes how to build your own Newton composites.

Newton provides some Ant tasks that you should import into your project. To use these Ant tasks you will need a binary install of Newton, which you can either download or build from the Newton source.

4.1.1. Ant Configuration

The Newton Ant tasks require some external jars on Ant's library path.

These can be specified explicitly each time ant is invoked, using `ant -lib`, but it's generally more convenient to use the antrc file:

4.1.1.1. Unix

Set environment variable NEWTON_HOME to point to the top of the Newton binary installation:

```
export NEWTON_HOME=$HOME/newton-1.2.x
```

Add the following to $HOME/.antrc:

```
ANT_ARGS="-lib $NEWTON_HOME/sdk/ant/lib"
```

4.1.1.2. Windows

The setup is similar, but you need very specific use of double quotes if the NEWTON_HOME path contains spaces (e.g. C:\Documents and Settings\user).

Set HOME environment variable so ant.bat finds antrc_pre.bat:

```
set HOME=%HOMEDRIVE%%HOMEPATH%
```

Set environment variable NEWTON_HOME to point to the top of the Newton binary installation:

```
set NEWTON_HOME="%HOME%\newton-1.2.x"
```

Add the following to %HOME%\antrc_pre.bat:

```
set ANT_ARGS=-lib %NEWTON_HOME%\sdk\ant\lib
```

Note:

If ant complains "Target 'and' does not exist in this project", then it is probably due to not using double quotes exactly as above - ant.bat is expanding %ANT_ARGS% and breaking on the 'and' in C:\Documents and Settings.

4.1.2. Composite build.xml

The following example shows how to build a component bundle named mycomposite that exports a single composite template:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<project name="mycomponent" basedir=".">
  <property environment="env"/>
  <property name="container.dir" location="${env.NEWTON_HOME}" />
  <!-- Note: container.dir must be set, as it is used in imported file
-->
  <import file="${container.dir}/sdk/ant/component-utils.xml" />

  <target name="compile">
      <!-- compile your source here -->
  </target>

  <target name="build" depends="compile" >
    <component.bundle name="mycomposite" version="1.0"
templates="mytemplate.xml">
      <exportpackage name="com.company.application.component.api"
version="1.0" />

      <fileset dir="${build.dir}/classes">
        <include name="com/company/application/component/**/*.class" />
      </fileset>

      <fileset dir="templates">
        <include name="mytemplate.xml" />
      </fileset>
    </component.bundle>
  </target>
</project>
```

See the existing component bundle Ant targets in the examples, for further details.

4.2. Newton: Creating composites - Hello World

This page describes how to create a simple Newton composite, HelloWorld. Before you start, you should configure Ant for Building Composites.

Note:

This article is not intended to introduce you to Newton composites in general. For this see the examples. The emphasis here is on writing your own composites.

Whilst the following is presented as a *hands-on* example of how component build process works, all of the code shown is also included in the Newton examples.

We will be creating two separate composites, one exposing the HelloWorld service and the other providing a command line extension called PrintHello which is used to test the HelloWorld service.

The HelloWorld interface looks like this:

```
package example.api;

public interface HelloWorld {
    String getGreeting();
}
```

Here is its implementation

```
package example.impl;

import example.api.HelloWorld;

public class HelloWorldImpl implements HelloWorld {
    public String getGreeting() {
        return "Hello World";
    }
}
```

The PrintHello class looks like this:

```
package example.cli;

import java.io.PrintWriter;

import org.cauldron.newton.command.CommandException;
import org.cauldron.newton.command.CommandRequest;
import org.cauldron.newton.command.CommandResponse;
import org.cauldron.newton.command.console.ConsoleCommandHandler;
import example.api.HelloWorld;

public class PrintHello extends ConsoleCommandHandler {

    private HelloWorld hello;

    public PrintHello() {
        super("printhello");
    }
```

```
    public void addHello(HelloWorld hello) {
        this.hello = hello;
    }

    public void removeHello(HelloWorld hello) {
        this.hello = null;
    }

    public void handle(CommandRequest request, CommandResponse response)
throws CommandException {
        PrintWriter out = getOut(response);
        out.println(hello.getGreeting());
    }
}
```

We will now create two composite templates to wrap components based on the classes shown above.

Firstly the helloWorld template:

```
<?xml version="1.0"?>
<composite name="helloWorld">
  <description>An example hello world composite</description>

  <service name="hello-export">
    <interface.java
interface="org.cauldron.newton.example.helloworld.api.HelloWorld"/>
    <binding.osgi/>
  </service>

  <component name="hello">
    <implementation.java.callback
impl="org.cauldron.newton.example.helloworld.impl.HelloWorldImpl"/>
  </component>

  <wire>
    <source.uri>hello</source.uri>
    <target.uri>hello-export</target.uri>
  </wire>
</composite>
```

And secondly the Print Hello template:

```
<?xml version="1.0"?>
<composite name="printHello">
  <description>An example hello world component</description>

  <reference name="hello" multiplicity="1..1">
    <interface.java
interface="org.cauldron.newton.example.helloworld.api.HelloWorld"/>
    <binding.osgi filter=""/>
  </reference>

  <service name="print-export">
    <interface.java
interface="org.cauldron.newton.command.console.ConsoleCommandHandler"/>
    <binding.osgi/>
  </service>
```

```
  <component name="print">
    <reference name="hello"/>
    <implementation.java.callback
impl="org.cauldron.newton.example.helloworld.impl.PrintHello"/>
  </component>

  <wire>
    <source.uri>hello</source.uri>
    <target.uri>print/hello</target.uri>
  </wire>

  <wire>
    <source.uri>print</source.uri>
    <target.uri>print-export</target.uri>
  </wire>
</composite>
```

Save each of these xml documents in the files `xml/helloworld-template.xml` and `xml/printhello-template.xml` respectively.

We now need to build our composites into bundles so that they can be deployed by Newton. Here is a Buildfile that achieves this:

Note:

Newton uses bld to generate a maven pom.xml from a Buildfile.

```
# helloworld
id:             helloworld
archetype:      NewtonBundle

tmpl.dir = xml
pkg = ${example.pkg}.helloworld

-depends:\
 command;org.cauldron.newton

-bundles:\
 api,\
 main,\
 cli

# api
api;name:       helloworld-api-bundle
api;-exports:   ${pkg}.api

# main
main;name:      helloworld-bundle
main;-exports:  ${pkg}.impl
main;-templates:\
 ${tmpl.dir}/helloworld-template.xml

# cli
cli;name:       printhello-bundle
cli;-contents:  ${pkg}.cli.*
cli;-templates:\
 ${tmpl.dir}/printhello-template.xml
```

```
# end
```

Having built the components we are almost ready to deploy them to the newton framework and test them out. In order to deploy the components we will need to create two further xml documents that describe an instance of each template.

```
<?xml version="1.0"?>
<composite name="helloworld">
  <bundle.root bundle="helloworld-bundle" version="1.0"/>
  <include name="helloWorld"/>
</composite>

<?xml version="1.0"?>
<composite name="helloworld">
  <bundle.root bundle="printhello-bundle" version="1.0"/>
  <include name="printHello"/>
</composite>
```

Save each of these composite documents in the files `xml/helloworld.composite` and `xml/printhello.composite` respectively. The build copies these files to `target/xml` and replaces the ${VERSION} tags.

Now we're ready, so startup the Newton framework and type in the following commands.

```
$ bin/container
> cds scan boot examples/helloworld/target
> installer install examples/helloworld/target/xml/printhello.composite
> installer install examples/helloworld/target/xml/helloworld.composite
> printhello
Hello World
```

4.3. Newton: CDS Protocol for External Clients

4.3.1. Intended Audience

Clients interacting with remote objects need to be familiar with the cds protocol in an environment comprising Newton containers since codebase annotations will be in this form.

4.3.2. Purpose

4.3.3. Prerequisites

4.3.4. Steps

Here is how to proceed.

4.3.4.1. Enable Server-side external CDS

For the time being this needs to be enabled manually. In a subsequent revision at least one container in the environment will fulfill this requirement automatically.

Assumption

You are in $Newton_HOME/install/Newton within the directory structure created by unzipping the download such that you were able to start the container by issuing bin/Newton.sh or bin/Newton.bat

```
> cds scan boot core/lib/components
> installer install file:core/instances/beacon-instance.xml
> installer install file:core/instances/cds-external-instance.xml
```

4.3.4.2. Configure Server-side external CDS

Configurable properties include:

- A multicast address and port used to allow clients to discover cds servers (mcast.addr and mcast.port)
- A scoping name which allows re-use of multicast addresses and ports (mcast.scope)

Configuration can be applied in a number of ways with preferential overrides. In order of decreasing preference:

1. The cds-external instance file located at $Newton_HOME/install/Newton/core/instances/cds-external-instance.xml (this is the expected source of server side configuration).

```
<?xml version="1.0"?>
<composite name="cds-external">
```

```
  <description>CDS External</description>
  <bundle.root bundle="cds-external" version="0.1"/>
  <include name="cds-external-template"/>
  <property name="mcast.port" value="2028" type="integer"/>
  <property name="mcast.addr" value="228.192.0.2" type="string"/>
  <property name="mcast.scope"
value="${system#Newton.fabric.name:-${system#user.name}}"
type="string"/>
</composite>
```

2. mcast.scope is determined from the system property newton.fabric.name (in the absence of equivalent instance configuration)
3. system properties:
   - org.cauldron.newton.net.channel.mcast.addr
   - org.cauldron.newton.net.channel.mcast.port
   - org.cauldron.newton.net.channel.mcast.scope
4. mcast.scope is determined from the system property user.name

- default mcast.addr is 228.192.0.2
- default mcast.port is 2028

4.3.4.3. Enable Client-side external CDS

We provide a jar which contains a custom URLStreamHandler capable of handling cds URLs. The jar is located at $Newton_HOME/install/newton/core/lib/bundles/cds-external-client.jar and should be included in your classpath along with log4j.jar. The property java.protocol.handler.pkgs=org.cauldron.newton.net.protocol enables the URLStreamHandler.

4.3.4.4. Configure Client-side external CDS

In order of decreasing preference:

1. mcast.scope is determined from the system property newton.fabric.name
2. system properties:
   - org.cauldron.newton.net.channel.mcast.addr
   - org.cauldron.newton.net.channel.mcast.port
   - org.cauldron.newton.net.channel.mcast.scope
3. mcast.scope is determined from the system property user.name

- default mcast.addr is 228.192.0.2
- default mcast.port is 2028

4.3.4.5. Example - The Jini Browser as an External CDS Client

Security Policy
A permissive policy named all.policy is required in the directory where this command is executed

Codebase Quotes on Unix / Linux

The codebase needs to be quoted as below to avoid it being interpreted as a series of commands

Additionally the Jini Lookup Service needs to be installed:

```
  > installer install file:core/instances/reggie-instance.xml


java -cp .:$JINI_HOME/lib/browser.jar:\
$LOG4J_HOME/lib/log4j.jar:$CONTAINER_HOME/core/lib/bundles/cds-external-client.jar:\
$CONTAINER_HOME/core/lib/app/if-jini-dl.jar:\
$CONTAINER_HOME/core/lib/app/if-jini-boot.jar \
-Djava.protocol.handler.pkgs=org.cauldron.newton.net.protocol \
-Djava.security.policy=all.policy \
-Djava.rmi.server.codebase="cds://jinitsk/browser-dl.jar?zone=boot&type=jini.codebas
\
com.sun.jini.example.browser.Browser
```

4.4. Newton: Connecting external Jini services

4.4.1. Overiew

In order for external Jini services to communicate with services launched from within Newton it is necessary for them to be able to connect to CDS urls (which are used to mark the rmi codebase on services exported from Newton).

The following example demonstrates how to extend the Chainlink Demo to incorporate an external (non Newton) link in the chain.

4.4.2. Example

First start Newton and launch reggie

```
$ newton.sh
...
> installer install core/instances/reggie-instance.xml
> installer install core/instances/server-cds-instance.xml
```

Publish demo lib the jars to the cds server

```
cds scan remote demo/build/lib
```

The following two lines start components that allow external clients to interpret cds urls - the first is a component that proxies cds requests over http. The second is a multicast beacon that allows external clients to discover where this http proxy is located.

```
installer install core/instances/cds-external-instance.xml
installer install core/instances/beacon-instance.xml
```

Finally start the chainlink components a and c and the cli as per the default chainlink demo

```
installer install demo/xml/chainlink/chainlink-jini-a.xml
```

```
installer install demo/xml/chainlink/chainlink-jini-c.xml
installer install demo/xml/chainlink/chainlink-jini-cli-a.xml
```

Now from the command line do the following

```
cd $newton_install/demo/resources/chainlink
sh external-chainlink.sh
```

(or windows)

```
cd %CONTAINER_INSTALL%\demo\resources\chainlink
external-chainlink.bat
```

When you type <code>jinichain chain</code> from within Newton you should now see that the b component has been provided by the external jvm launched from the script.

4.4.3. Discussion

Within the external-chainlink scripts we have referenced the property:

```
-Djava.protocol.handler.pkgs=org.cauldron.newton.net.protocol
```

This plugs in a Java protocol handler which communicates with the http bridge from the jar file

```
$NEWTON_HOME/lib/core/bundles/cds-external-client.jar
```

This and the lib

```
$NEWTON_HOME/lib/core/app/if-jini-boot.jar
```

are the only things that need to be added to an external jini client in order to allow them to intercommunicate with components instantiated within Newton.

For the cds beacon the default multicast group is 228.192.0.2 and the port is 2028. These defaults can be overridden if need be by setting system properties org.cauldron.newton.net.channel.mcast.addr and org.cauldron.newton.net.channel.mcast.port.

4.5. Newton: Building from Source

This page describes how to build Newton from source code.

The latest source code can be checked out from the Newton source repository, or a source distribution can be obtained from the Downloads page.

4.5.1. Software required to build Newton

The software required to build Newton is summarized in the table below:

Software	Download URL
Apache Ant 1.7.0 or later	http://ant.apache.org/bindownload.cgi
Apache Maven 2.0.8 or later	http://maven.apache.org/download.html
Java JDK 5 or 6	http://java.sun.com/javase/downloads/

4.5.2. Maven Preparation

The Newton build requires the local Maven repository to be initialised:

```
cd src
mvn -f assemble/prepare/pom.xml package
```

Note:

You only need to do this once, or after you destroy ~/.m2/repository.

4.5.3. Build targets

To build Newton, change directory to the top of the Newton tree, then run:

```
ant
```

This creates a full distribution, including the examples, in the `src/target/newton-dist.dir` directory.

Note:

See src/README for details of running Maven goals directly.

5. Tools

5.1. Newton: Troubleshooting Multicast

5.1.1. Multicast in Newton

Newton makes use of multicast for Jini discovery

1. For Jini discovery - This is how Newton components in different containers locate the jini registry, and hence each other

```
224.0.1.84 jini-announcement - port 4160
224.0.1.85 jini-request - port 4160
```

To support this your network must propagate multicast packets between all of the Newton instances you want to see each other. This will typically be all instances on a particular subnet or switch. If you find that your Newton containers are unable to see each other you should check that you have multicast configured appropriately.

Note that even when you are using a single machine there are configurations in which multicast packets are not propagated between processes.

The following sections describe the multicast diagnostics tool included in the Newton distribution, as well as things to check if multicast isn't working.

5.1.2. Testing Multicast with the NicNack tool.

The Newton distribution includes a lightweight command line tool called NicNack that can be used to see if multicast packets are travelling between two hosts or not. NicNack is network interface aware, i.e. it provides separate information for each network interface.

NickNac can be run using the nicnac.sh (unix) or nicnac.bat (windows) which can be found in the NEWTON_HOME/bin directory. For windows users this file must be run from this directory or NEWTON_HOME must be set.

NicNack runs in three modes:

1) List mode. This just lists all network interfaces on the machine, along with ip and host name information. For example:

```
sh bin/nicnack.sh list
NICs:

display name: vmnet8
name: vmnet8
ipv6: fe80:0:0:0:250:56ff:fec0:8
ip: 172.16.206.1

display name: vmnet1
name: vmnet1
ipv6: fe80:0:0:0:250:56ff:fec0:1
ip: 192.168.32.1
```

```
display name: wlan0
name: wlan0
ipv6: fe80:0:0:0:211:50ff:fe1b:d845
ip: 192.168.2.2

display name: lov
name: lo
ipv6: 0:0:0:0:0:0:0:1
ip: 127.0.0.1 -> localhost
```

2) Send mode. In this mode nicnack multicasts a custom message every two seconds to a specified multicast group and port. This can be read by other NicNack instances in receive mode.

```
> sh bin/nicnack.sh send 225.123.123.123 12345 hello
vmnet8: preparing
vmnet1: preparing
lo: preparing
wlan0: preparing
vmnet8: configured socket
vmnet1: configured socket
wlan0: configured socket
vmnet8: sending
vmnet1: sending
vmnet1: sent hello
wlan0: sending
wlan0: sent hello
vmnet8: sent hello
lo: configured socket
lo: sending
lo: sent hello
vmnet1: sent hello
vmnet8: sent hello
lo: sent hello
wlan0: sent hello
vmnet1: sent hello
wlan0: sent hello
vmnet8: sent hello
lo: sent hello
```

3) Receive mode. In this mode NicNack listens for and displays messages sent to a specified multicast group and port by NicNack instances in send mode. Used in conjunction with Send mode this can be used to establish whether or not multicast traffic is successfully propagated between two hosts. Note that multicast visibility is not symmetric, i.e. jhost A's ability to send multicast packets to host B does not imply host B's ability to send packets to host A. A sample from a receive mode NicNack session follows.

```
> sh bin/nicnack.sh receive 225.123.123.123 12345
vmnet8: preparing
lo: preparing
wlan0: preparing
vmnet1: preparing
wlan0: configued socket
wlan0: receiving
vmnet1: configued socket
vmnet1: receiving
```

```
vmnet8: configued socket
vmnet8: receiving
lo: configued socket
lo: receiving
wlan0: received 'hello'
vmnet1: received 'hello'
vmnet8: received 'hello'
lo: received 'hello'
```

5.1.3. Troubleshooting

If, having run the NicNack utility, you suspect that IP multicast may be the issue, the following two areas should be looked at in more detail.

5.1.4. Firewalls

The security firewall on one, a subset, or all of your machines that are running the Newton environment may be configured by default to block IP multicast traffic.

Linux - To enable multicast send / receive capability for Linux systems, insert the following entry into the operating system's iptable

```
INPUT -d 224.0.0.0/4 -j ACCEPT
```

Windows - In the case of Windows XP, by default, the Group Policy settings for the Windows Firewall are "Not Configured" for all objects. This allows the Windows Firewall to use its default settings, which are quite restrictive. With respect to *multicast* the default settings prohibits unicast response to multicast or broadcast requests.

On the relevant machines, edit *Network* > *Network Connections* > Firewall and set a disable policy

```
Prohibit unicast response to multicast or broadcast requests

Not Configured - The incoming unicast response is accepted if received
within 3 seconds. The setting can be overridden by a local
administrator.

Enabled - The incoming unicast response is dropped. This cannot be
overridden by a local administrator.

Disabled - The incoming unicast response is accepted if received within
3 seconds. This cannot be overridden by a local administrator.
```

For other Firewall products or alternative Microsoft operating system versions. Check relevant documentation.

5.1.5. Network Configuration

Simple layer II network switches treat *multicast* traffic in the same manner as *broadcast* traffic, that is, they will forwarded multicast packets to all active switch ports. If your Newton test machines connect to a layer II network, comprising of one or more simple layer II ethernet switches (these interconnected without intervening layer III routers), then the network is unlikely to be the cause of IP multicast connectivity issues.

In more sophisticated environments, network infrastructure supports a mulitcast protocol known as IGMP. Within an IGMP enabled network environment, traffic associated with a multicast group is only forwarded to ports that have members participating in that group. A layer-2 switch supporting *IGMP snooping* can passively snoop on IGMP Query, Report and Leave (IGMP version 2) packets transferred between IP Multicast Routers/Switches and IP Multicast hosts (on each switch port), to learn the required IP Multicast group membership required by each port. The advantage of using *IGMP snooping* is that it generates no additional network traffic, whilst significantly reduce multicast traffic passing through your switch - as all multicast is only targeted to hosts that have registered interest in the multicast group.

If Newton functions correctly when run on a single machine and also when run in a distributed environment with machines connected via a simple layer II network, but fail in a more complex network environments, then multicast configuration of the network is the the most likely cause. In such circumstances politely explain the problem to your network administrators. The network administrators will be able to help you diagnose the issue in greater detail, and may be willing/able to disable *IGMP snooping* on the relevant switches to verify whether or not IGMP is a contributing factor.

6. Reference

6.1. Spring OSGi in Newton reference

6.1.1. Introduction

Newton's support for Spring Dynamic modules for OSGi provides the following capabilities

- The ability to treat Spring-OSGi bundles as SCA composites and thereby use them just like any other Newton composite.
- The ability to promote OSGi level services and references to SCA level, so that they can interact using SCA bindings other than OSGi
- The ability to source the properties in Spring-OSGi configuration files from the SCA properties in their SCA composite document. This makes it easy to deploy and configure Spring-OSGi bundles in a single step.

The approach taken by Newton leaves Spring-OSGi components oblivious to the fact that they are interacting with anything other than the OSGi registry, so their normal lifecycle is undisturbed.

6.1.1.1. Limitations

At present Newton's Spring-OSGi support is only available when running Newton under Equinox, its default OSGi framework. Support for Felix and Knopflerfish is planned.

6.1.2. Using Spring OSGi bundles with Newton

6.1.2.1. Configuration basics

To use Spring-OSGi bundles with Newton two new Manifest entries should be added

```
Installable-Component: true
Installable-Component-Templates=<path to templates>
```

The first of these simply identifies the bundle as a Newton bundle, and the second tells Newton where to look for the SCA template file (in the case of Spring-OSGi you'll only need one template). Best practice when using Spring-OSGi bundles is to put templates in the META-INF/newton directory alongside the META-INF/spring directory holding Spring configuration files.

To see how these entries can be inserted using Spring-OSGi's build system, take a look at one of the examples (http://examples/) .

Spring OSGi based templates can only contain a single SCA level component. This should have implementation type *implementaton.spring*, in the namespace

http://newton.cauldron.org/springdm. *implementaton.spring* is an implementation type that knows how to wrap a Spring OSGi bundle. It has no attributes because the bundle on which it is based is already known at composite level. Here's a minimal example

```
<composite name="MyComposite"
xmlns:sdm="http://newton.cauldron.org/springdm">
    <component name="MyComponent">
         <sdm:implementation.spring />
    </component>
</composite>
```

6.1.2.2. Promoting References and Services from OSGi to SCA

Newton can promote OSGi level services and references to SCA level, where they can be assigned SCA level bindings, and thereby interact with non-OSGi services, e.g. remote ones.

To tell Newton that a service should be promoted to SCA level, the service should be given a single service property called newton.sca.service, the value of which will be used as the service name of the promoted service on the corresponding implementation.spring based SCA component. Here's an example

From bundle-context-osgi.xml

```
<osgi:service ref="foo" interface="org.cauldron.newton.Foo">
    <service-properties>
        <entry key="newton.sca.service" value="promotedFoo"/>
    </service-properties>
</osgi:service>
```

This promotes the foo service, giving it the SCA level name promotedFoo. In the corresponding Newton template file this is used as below, and can be wired to any matching SCA composite level service

```
<component name="MyComponent"
xmlns:sdm="http://newton.cauldron.org/springdm">
  <service name="promotedFoo" />
  <sdm:implementation.spring />
</component>
```

Similarly, to tell Newton that a reference should be promoted to SCA level it should be given a filter of the form (newton.sca.reference=<SCA level name>).The right hand side of which is used as the name for the SCA reference on the SCA component implemented by Spring. Here's an example:

```
<osgi:reference id="bar" interface="org.cauldron.newton.Bar"
                         filter='(newton.sca.reference=promotedBar)'
                         cardinality="0..1">
    <osgi:listener bind-method="bindBar" unbind-method="unbindBar"
ref="BarBean"/>
</osgi:reference>
```

This promotes the bar reference, giving it SCA level name promotedBar. In the corresponding template file this is used as below, and can be wired to any matching SCA composite level reference

```
<component name="MyComponent"
```

```
http://newton.cauldron.org/springdm">
  <reference name="promotedBar" />
  <sdm:implementation.spring />
</component>
```

6.1.2.3. Configuring Spring OSGi bundles using SCA properties

Spring-OSGi is able to source bean values from the OSGi Configuration Admin service. Newton is able to source Configuration Admin properties from SCA properties. Consequently Newton SCA properties can be used to set Spring bean properties. To see how this works, take a look at the following example.

On the Spring side, properties are sourced from the Configuration Admin service. For full details see the Spring OSGi documentation. (http://static.springframework.org/osgi/docs/1.0.1/reference/html/appendix-compendium.html#compendi Omitting namespaces, here's a minimal example:

```
<beans
   <property-placeholder persistent-id="org.cauldron.newton.people"/>
   <bean name="Person" class="org.cauldron.newton.people.PersonImpl" >
            property name="name" value="${person.name}"/>
   </bean>
</beans>
```

Here the persistent-id is the Configuration Admin service pid (persistent id) from which the bean properties are being sourced.

On the Newton side, any implementation.spring based SCA component that sets the property *service.pid* will have all of its other properties passed to the Configuration Admin service using this service pid. So, for example, to set the property in the above beans file a minimal SCA document would be.

```
<composite name="springlink"
xmlns:sdm="http://newton.cauldron.org/springdm">
  <component name="springLink">
    <property name="service.pid" value="org.cauldron.newton.people"
type="string" />
    <property name="person.name" value="Fred" type="string" />
    <sdm:implementation.spring />
  </component>
</composite>
```

Note that the properties in Newton template documents can be overridden and augmented in composite instance documents. This leads to a useful design pattern in which the service.pid and default configuration properties are set in the template and any custom values (but not the service pid) are set by overridding these in the instance documents.

6.1.2.4. Configuring different Spring-OSGi versions

By default Release 1.2 of Newton supports Spring-OSGi version 1.0.1. This version number can be changed by setting the container.springDMVersion property in the the <Newton_HOME>/etc/config.ini.

To simplify loading of the required Spring-OSGi bundles into CDS the following Newton

publish file, which loads a minimal set, may be useful

```
<?xml version="1.0" encoding="iso-8859-1"?>
<publish>
    <item name="spring-osgi-core.jar"
path="dist/spring-osgi-core-1.0.1.jar" />
    <item name="spring-osgi-extender.jar"
          path="dist/spring-osgi-extender-1.0.1.jar" />
    <item name="spring-osgi-io.jar" path="dist/spring-osgi-io-1.0.1.jar"
/>
    <item name="spring-core-2.5.1.jar" path="lib/spring-core-2.5.1.jar"
/>
    <item name="aopalliance.osgi-1.0-SNAPSHOT.jar"
          path="lib/aopalliance.osgi-1.0-SNAPSHOT.jar" />
    <item name="jcl104-over-slf4j-1.4.3.jar"
          path="lib/jcl104-over-slf4j-1.4.3.jar" />
    <item name="slf4j-api-1.4.3.jar" path="lib/slf4j-api-1.4.3.jar" />
    <item name="slf4j-log4j12-1.4.3.jar"
path="lib/slf4j-log4j12-1.4.3.jar" />
    <item name="spring-aop-2.5.1.jar"  path="lib/spring-aop-2.5.1.jar"
/>
    <item name="spring-beans-2.5.1.jar"
path="lib/spring-beans-2.5.1.jar" />
    <item name="spring-context-2.5.1.jar"
path="lib/spring-context-2.5.1.jar" />
</publish>
```

To use this file simple save it as publish-spring-osgi-1.0.1.xml and load from the Newton command line using

```
cds publish <zone> <path to publish file> <Spring-OSGi install
directory>
```

To use this with later versions of Spring-OSGi the version numbers (at least) will have to be updated. Any more substantial changes will be reflected in a later version of this document.

6.2. Composite Instance Reference

As well as packaging SCA composites in bundles, Newton also allows developers to create “instance” documents which point to an existing SCA composite and (potentially) modify it in some way.

This allows the original software developer to specify the basic behaviour of an SCA composite in their bundles, but for another developer (acting as a user of the software) to customise the composite for his/her environment.

6.2.1. XML Reference

The following example creates an instance of a composite with the named “helloWorld” from the bundle with the bundle-symbolic-name “org.example.helloworld” and bundle version “1.0”:

```
<?xml version="1.0"?>
<composite name="helloWorld-instance">
```

```
  <bundle.root bundle="org.example.helloworld" version="1.0"/>
  <include name="helloWorld"/>
</composite>
```

The bundle root element specifies the bundle that supplies the composite file and provides the classpath (via local references or import statements) for all classes loaded by this composite.

The following is an example of how the helloWorld composite may look:

```
<?xml version="1.0"?>
<composite name="helloWorld">
  <property name=”defaultName” value=”world” />
  <service name=”api”>
    <interface.java interface=”org.example.HelloWorld” />
    <binding.osgi />
  </service>
  <component name=”impl”>
    <implementation.java impl=”org.example.impl.HelloWorldImpl” />
  </component>
  <wire>
    <source.uri>api</source.uri>
    <target.uri>impl</target.uri>
  </wire>
</composite>
```

To follow the original example we would then package this composite file in the META-INF folder of a bundle with the following manifest headers:

```
Manifest-Version: 1.0
Bundle-SymbolicName: org.example.helloworld
Bundle-Version: 1.0
Bundle-Name: Hello World
Installable-Component: true
Installable-Component-Templates: META-INF/helloWorld.composite
```

Note:

The Installable-Component and Installable-Component-Templates manifest headers are required to indicate to Newton firstly that the bundle contains templates and secondly the location of the templates within the bundle. The templates header can contain a comma separated list of paths if more than one composite is provided by a bundle.

To complete the picture we could then create a second instance which customises the hello world composite to publish it's interface remotely and change the property default name to something else:

```
<?xml version="1.0"?>
<composite name="helloWorld-instance">
  <bundle.root bundle="org.example.helloworld" version="1.0"/>
  <include name="helloWorld"/>
  <property name=”defaultName” value=”Dave” />
  <service name=”api>
    <binding.rmi />
  </service>
</composite>
```