

OSGi in Distributed Robotic Systems

Ben Coughlan
Australian National University
Canberra, Australia

`ben.coughlan@anu.edu.au`

June 19, 2009

Abstract

Distributed computing mechanisms have been a research topic since object oriented languages first appeared. There are, however a number of issues which prevent these mechanisms from allowing developers to build distributed applications as easily as they would like.

A truly transparent and dynamic distribution layer could be used to great affect in facilitating the interactions of autonomous agents. When provided with an ubiquitous and self managing network infrastructure, such agents can provide services to each other and seamlessly collaborate with minimal planning from the developers.

The modularity paradigm of OSGi provides researchers with a new approach to software distribution. A library called Remote OSGi allows modules to be arbitrarily distributed among a collection of nodes by replicating their service level interfaces and supplying an implementation consisting of remote procedure calls.

This project is focused on combining the peer-to-peer protocols defined in the JXTA standard with R-OSGi to provide a robust and pervasive network of agents. The task is to simply implement a transport layer interface with JXTA.

A lot of emphasis is placed on the ability to build OSGi bundles with one click. There are a number of tool available to help with OSGi development. The focus of this project is Maven and its plugins.

Java based robotics also benefits from direct access to hardware via the Java Native Interface. OSGi support for the JNI is investigated and techniques for development of JNI inclusive bundles are presented.

A number of scaleability issues were discovered with this approach and time constraints prevented the implementation of the alternative. However the alternative design is explained and descriptions of a competing method are provided.

Acknowledgments

Thanks to Jan Rellermeyer and his team at ETH in Zurich for their fantastic work in the area, and to Jan for personally taking the time to answer my questions. I am truly impressed by your ideas and look forward to seeing your future contributions.

Thank you to Shayne Flint for his unquestioning faith in me despite strong evidence that it may have been misplaced.

Contents

| | | |
|----------|-------------------------------------------------------|-----------|
| 1 | Introduction | 5 |
| 1.1 | Overview | 5 |
| 1.2 | Motivation | 6 |
| 1.3 | Objective | 7 |
| 2 | Background | 9 |
| 2.1 | Distributed Computing | 9 |
| 2.2 | OSGi | 11 |
| 2.3 | Remote OSGi | 12 |
| 2.4 | JXTA | 13 |
| 3 | Planned Approach | 15 |
| 3.1 | Combining R-OSGi and JXTA | 15 |
| 3.1.1 | R-OSGi Network Channels | 15 |
| 3.1.2 | JXTA Network | 17 |
| 3.1.3 | JXTA R-OSGi Bridge | 18 |
| 3.2 | Developing and Deploying OSGi bundles | 18 |
| 3.3 | Including Native Code within an OSGi Bundle | 20 |
| 4 | Results | 21 |

| | | |
|----------|--------------------------------------------------------|-----------|
| 4.1 | Developing OSGi bundles | 21 |
| 4.2 | Scaleability Issues | 23 |
| 4.3 | Preserving Language Semantics | 24 |
| 4.4 | OSGi with Native Libraries | 25 |
| 5 | Future Work | 27 |
| 5.1 | Mapping OSGi services to JXTA advertisements | 27 |
| 6 | Conclusion | 31 |

Chapter 1

Introduction

1.1 Overview

According to a team from Sun Microsystems, the idea of leveraging the interfaces found in object oriented software design as a means of distributing an application is not new [1], in fact it is a recurring theme for researchers every decade or so. The goal is usually to find a way of transparently distributing the objects within an application so that decisions about where the code is executed can be deferred until the application is deployed and even changed while it is running.

The authors of [1] refer to this as an attempt to unify access to objects residing on the local system with objects on the other side of some distribution medium, i.e. a network. They conclude with strong arguments against such a unification, stating that the differences between local and remote object are irreconcilable and organisations would be wise to allocate their research & engineering resources elsewhere.

Efforts towards explicit software modularisation, beyond what is provided by object orientation, lead to the forming of the Open Services Gateway Initiative, now known as the OSGi Alliance. The OSGi Alliance authors and maintains a standard ([2]) for a modularisation framework which defines mechanisms for deploying and managing independent software modules. This concept provides a new approach to transparent software distribution that overcomes some of the issues in a purely object oriented approach.

This idea has been investigated and demonstrated by researchers at the Swiss Federal Institute of Technology (ETH) in Zurich, resulting in the development of Remote-OSGi (R-OSGi) [3]. R-OSGi takes advantage of the forced separation of a service and its interface by creating ‘proxy’ services which replace a service with a layer of remote procedure calls. The aim of R-OSGi is to transparently produce proxy services in a remote instance of an OSGi framework, allowing distributed modules to interact seamlessly.

There has been some investigation into using OSGi to support devices in home networks [4] and virtual home environments [5], as well as a recent look at R-OSGi in cloud computing [6]. Each of these look towards a ubiquitous and heterogeneous network, where the devices on the network can interact and collaborate seamlessly. Hopefully the benefit of such a network in those environments is obvious and I shall leave it to the respective authors to demonstrate. This paper is concerned with the development of such a network for use in systems of autonomous agents, possibly with robotic capabilities.

For such a system to be effective and robust, it is desirable that it be decentralised and not rely on a server infrastructure. A peer-to-peer approach provides an attractive alternative. JXTA defines a collection of protocols for the complete implementation of a peer-to-peer application [7]. Its concepts are analogous to some found in the OSGi standard, making it quite suitable in this situation. This paper is mainly concerned with smoothly combining R-OSGi and JXTA while maintaining the benefits that each provides.

1.2 Motivation

Distributed robotic systems can be developed in a way such that the individual agents that make up the system are abstracted away and external interactions are experienced from the system as a whole. This approach is very powerful when agents can be developed in isolation with the required distribution mechanism ‘built-in’ even if there is no intention for them to be used in such a system. Automating the management of the underlying network and allowing for service discovery and even inter-agent routing allows agents to be developed and deployed into an arbitrary system with

significantly reduced effort.

When what is conceptually one agent is thought of as a collection of agents - being the individual components - the mechanism described above can also become a powerful tool for the modularisation of complex robots. As an example; consider a robot consisting of a mobile platform, a chassis with wheels etc. and some form of actuator like an arm with a gripper. The concerns of these two components are quite different; one controls the location of the robot while the other manipulates the environment at that location. These two components can be deployed separately to achieve individual goals in isolation, or they can be combined to achieve a common goal. If both act as individual agents and have a means of communication, there is a lot of flexibility regarding their deployment and potential collaborative efforts.

A system of agents is not always explicit. There are real-world examples of individual agents interacting/actively avoiding each other to achieve their own goals. The well known example being vehicles participating in traffic. Developers working on one agent are not expected to consider such interactions with arbitrary agents. They will likely focus on a known set of agents if any, and attempt all other interactions by sampling their environment and behaving appropriately. A ubiquitous network providing a standard communication mechanism provides a way for independent agents to implicitly collaborate more effectively than sampling the environment allows.

1.3 Objective

This project aims to combine R-OSGi and JXTA to form a middleware that allows for spontaneous collaboration between robotic agents.

The benefits of each technology are to be maintained wherever possible. This includes the *transparency* and *efficiency* of R-OSGi, as well as the *scalability*, *security* and *robustness* of JXTA, among others. These two technologies do clash, mostly with regard to the communication semantics. In these situations, trade-offs will be required.

The modularity of an OSGi application is to be observed. This requires the leveraging of interfaces already present in R-OSGi and JXTA as well as a properly considered design for areas that have not yet been developed. As JXTA does not currently have

an OSGi compatible implementation, services will need to be defined, as well as the semantics of managing the JXTA network.

Given Java suffers a separation from the hardware due to being run in a virtual machine, there will be some investigation into the Java Native Interface and how it is used from within an OSGi framework. This will allow hardware drivers implemented in native machine code to be accessed from within the framework.

There is also a requirement to investigate the security mechanisms provided by JXTA and OSGi. These can be used to implement a hierarchy of agents and provide a “chain of command” as well as authorisation services among groups of agents.

Chapter 2

Background

2.1 Distributed Computing

In 1994, a team from Sun Microsystems wrote a report discussing distributed computing research attempting to take advantage of the semantics of object orientation [1]. They expressed the vision researchers had, of unified objects capable of being arbitrarily distributed over a number of nodes, and where the method calls between local and remote objects were syntactically and semantically identical. This concept has been investigated many times, resulting in platforms such as the COMMON OBJECT REQUEST BROKER ARCHITECTURE¹ (CORBA). Developers using CORBA use an interface definition language to define the objects and methods calls which are then accessed through an object request broker. This approach is not native to any particular language, and is far from the vision of unified objects.

According to [1], research in this area occurs approximately every decade or so and usually results in one of two outcomes; one emphasising integration with a current language model, while the other emphasises solutions to problems inherent in distributed computing. The authors identify four issues they consider to be irreconcilable between local and remote objects.

¹CORBA - <http://www.corba.org>

Latency is the most obvious hurdle faced with distributed computing. It is also the most difficult to overcome, as the physical limits of a network connection will never come close to those of purely local hardware [1]. This leaves developers of a distributed application with the need to consider the implications of a remote call versus a local one, removing all transparency from the distribution mechanism. The only alternative is to suffer the arbitrary performance hits incurred with the possibility that all method invocations can be remote.

Concurrency is of course the idea that multiple ‘threads’ of execution can be running simultaneously. When an application is running in a distributed environment, it has the opportunity to experience truly asynchronous behaviour. All developers should be familiar with this concept, and most modern languages include mechanisms for controlling the behaviour of concurrent systems. However the semantics of concurrency in a system running on a single operating system above a single processor, are quite different to those of a system spanning multiple platforms.

Partial Failure includes all the modes of failure that distribution adds to a system. A local call will usually result in an error being returned or an exception thrown in the event it cannot complete its task. Any other behaviour is likely the result of a catastrophic event in the system which can not be recovered. In a distributed environment, such events are considered ‘normal’. Nodes in the system may become unresponsive meaning that a remote method invocation may not return at all. Current platforms cannot handle these events transparently, requiring some knowledge of the distributed behaviour to recover.

Memory Access refers to where the objects are stored on physical hardware and how they are accessed by the system. This is usually limited to the platform’s physical RAM, or a virtual memory structure supported by an operating system. The need for synchronisation and the expectation of access latency orders of magnitude smaller than a network will make these concepts very difficult to achieve in a distributed system [1].

The main impact these issues have on the objectives outlined in 1.3 is that of trans-

parency. A framework that hides the remote nature of a method invocation from the developers is inherently dangerous as it cannot be expected to perform as if the calls were local.

The team from Sun went on to write the *javax.rmi* packages found in the standard Java API (from 1.1) [8]. This of course took all of the above issues into consideration and forms a useable and conceptually robust RMI layer. It does not however, allow for the distribution of systems that were not explicitly designed and developed to do so.

2.2 OSGi

The Open Services Gateway Initiative (OSGi) was started in 1999 and is now known as the OSGi Alliance [2]. The OSGi standard is now in its fourth iteration and has been implemented by various vendors. Currently there are four open source implementations available, APACHE FELIX, ECLIPSE EQUINOX, KNOPFLERFISH and CONCIERGE which only implements version 3 of the standard.

OSGi is a modular support framework for Java. It handles modules called *bundles* and provides functionality such as dependency tacking and resolution, bundle life-cycle management, and service registration and provision.

Bundles are standard Java archives (jars) with added metadata specifying package dependencies and the packages the bundle wishes to export to the framework. Each bundle installed in a framework is given a unique classloader. The classpath of a bundle is determined by the packages included in the archive, and the packages specified to be imported in the bundle's manifest. The bundle's classloader delegates the loading of imported packages to the classloader of the bundle exporting them. This completely defines the dependencies between bundles and encourages such coupling to be as loose as possible.

Bundles can also register *services* to the framework for consumption by other bundles. A service requires the separation of the interface from the implementation. Bundles wishing to consume a service request instances from the framework with the name of their interface. The OSGi standard allows for bundles to enter and leave a framework

spontaneously. To allow for this, the standard defines events that occur when services are registered or removed and allows bundles to observe these events.

2.3 Remote OSGi

Remote OSGi² (R-OSGi) was developed at the Federal Institute of Technology (ETH) in Zurich in 2005 [3]. Since its release its author has continued to develop it and use it as a topic of research into distributed computing. He has since been invited to participate in developing future versions of the OSGi specification.

The core of R-OSGi is installed as a single bundle within an OSGi framework. When it is activated, it registers a *ServiceTracker* to observe the framework instance and invoke callbacks when a service is registered for remote access. Service that are intended to be accessed remotely have the property *service.remote.registration* set to true upon registration with the OSGi framework.

When R-OSGi finds a service to be exported, it notifies its remote clients via a symmetric lease. The clients then fetch the service interface from the server, as well as any packages required to fulfil the service's dependencies on the client side. These are all transferred as Java byte-code and are reconstructed into a proxy bundle which is installed on the client side. Other bundles in the client framework see and interact with the proxy like any other service. All the methods in the service implementation are replaced with remote procedure calls. R-OSGi performs the byte-code analysis required to transfer the minimal set of implementation to create the remote proxy bundle.

Given the issues of scalability in a network with many nodes, R-OSGi avoids having all services proxied by employing a whiteboard pattern, where bundles can register their demand for a service as a *DiscoveryListener* [3]. The simple registration of a service for remote access constitutes a statement of supply. These statements of supply & demand are combined with a service discovery protocol which allow instances of R-OSGi to selectively distribute proxy bundles and bind with remote services.

R-OSGi also has provision for generating 'smart' proxies that supply some of the logic

²R-OSGi - <http://r-osgi.sourceforge.net/>

required to the client. This reduces the amount of network traffic and greatly increases the performance of invocations of simple methods in the proxy. These smart proxies are supplied by supplying an *abstract* implementation of the service in which a subset of the service interface is implemented. The byte-code of the implemented methods is transferred and recreated on the client, while the methods that are not implemented will be replaced with remote procedure calls.

2.4 JXTA

Peer-to-peer networking has been a popular method for implementing decentralised network topologies. Without the need for a dedicated server node, a network is free to be more flexible and robust. When nodes in a network are able to organise themselves independently, there is the opportunity for ad-hoc and spontaneous interaction between individual nodes.

JXTA is a standard [9] defining a collection of XML based protocols intended to provide the functionality required for peer nodes to manage their relationships, advertise their services and invoke services on others. JXTA goes further than a standard peer-to-peer implementation by also providing mechanisms for inter-peer routing, peer grouping, and even an authorisation method based on a public key infrastructure contained within the peer network.

The following protocols are provided in the complete JXTA implementation [7].

- Peer discovery protocol
- Peer resolver protocol
- Rendezvous protocol
- Peer information protocol
- Pipe binding protocol
- Endpoint routing protocol

Smaller implementations of JXTA can be developed by only including a subset of these protocols, much like the JXME implementation.

Every resource and service available on a JXTA network is described by an *Advertisement*. Advertisements can be published by pushing them to the node's local cache or broadcasting them to a peer groups where they can be consumed immediately, or saved in the remote nodes cache's. When a node wishes to bind to, and consume a service it attempts a service discovery. This discovery starts with a search of the local cache for advertisements before extending to other peers, possibly propagating over several subnets.

JXTA nodes can also route communications between nodes on separate networks. This allows distance nodes to interact with each other in the same way, as if they were on the same network. It also enables routing between different transports, for example a node on serial bus can send messages to nodes across the internet by routing them through another node, situated on a host connected to the serial bus and the network. Firewalls and NAT can also be traversed allowing transparent routing across such boundaries. JXTA provides a robust and pervasive layer for a decentralised communications platform.

Chapter 3

Planned Approach

3.1 Combining R-OSGi and JXTA

The main concept of this project is to provide a JXTA transport for the R-OSGi protocol. R-OSGi provides the semantics and byte-code manipulation features required to reproduce services across the network, while JXTA can provide the network management, routing and security mechanisms.

The design for this is quite simple. A bridging bundle will be produced to supply a `NetworkChannel` implementation for R-OSGi that uses a JXTA network (see section 3.1.1). The bridge will also supply an JXTA advertisement for R-OSGi to be published to the network, allowing other nodes to discover and consume it as a service.

3.1.1 R-OSGi Network Channels

Originally, R-OSGi was developed to only use TCP sockets for all communication. After seeing the need for other network interfaces such as bluetooth, R-OSGi has been extended to allow alternative implementations of a *network channel* [3]. A third party bundle can provide an implementation of the `NetworkChannelFactory` interface and register it with the OSGi framework. An additional property is supplied with the service registration to define the *URI scheme* that the network channel provides. This provides a non-invasive way to supply a new transport to the R-OSGi protocol.

```
public interface NetworkChannelFactory {
    NetworkChannel getConnection(final ChannelEndpoint endpoint,
                               final URI endpointURI) throws IOException;
    void activate(final Remoting remoting) throws IOException;
    void deactivate(final Remoting remoting) throws IOException;
    int getListeningPort(final String protocol);
}
```

Figure 3.1: NetworkChannelFactory interface definition

The interface definition of a `NetworkChannelFactory` can be seen in figure 3.1. It provides the methods used by R-OSGi to manage the life cycle of the factory (and the channels it provides). A bundle registers an implementation of a `NetworkChannelFactory` to the framework and sets the “protocol” property to the URI scheme the network channels will support, i.e. `http`, `tcp`. When R-OSGi connects to a remote peer it is first given a URI describing where it is and how to connect like `tcp://hostname:port`. R-OSGi will search the framework for all services implementing a `NetworkChannelFactory` and the find one whos *protocol* matches the scheme in the URI. The factory is activated and used to obtain a `NetworkChannel`.

```
public interface NetworkChannel {
    String getProtocol();
    URI getRemoteAddress();
    URI getLocalAddress();
    void bind(ChannelEndpoint endpoint);
    void close() throws IOException;
    void sendMessage(RemoteOSGiMessage message) throws IOException;
}
```

Figure 3.2: NetworkChannel interface definition

R-OSGi collects its input data by observing the framework for service registrations and providing the implementations of proxy bundles. This information is transformed into `RemoteOSGiMessage` objects before being handed to the `NetworkChannel`'s `sendMessage` method, which is expected to deliver it to the remote peer. The interface layers involved in this process are briefly illustrated in figure 3.3.

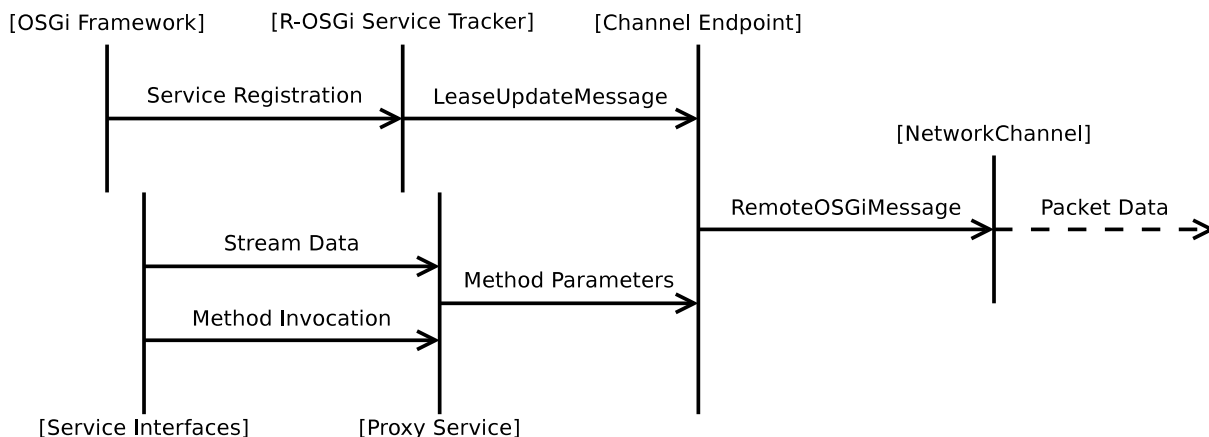


Figure 3.3: Default R-OSGi interface traversal

3.1.2 JXTA Network

The JXTA implementation for Java needs to be ported to an OSGi environment. This requires all external dependencies to be listed and, if needed, also ported to the OSGi environment. The nature of the current implementation is quite monolithic. As it is a reference implementation, it is trying to demonstrate all facets of the JXTA standard. This makes porting it to OSGi a significant task which is difficult to estimate given the large number of dependencies involved.

The JXTA bundle's behaviour must also be appropriate. A `BundleActivator` must be implemented to register the appropriate services with the OSGi framework. The activator is also responsible for starting the network instance and providing mechanisms to respond to any events that occur.

By keeping the JXTA network isolated in its own bundle, other applications in the OSGi framework will be able to interact with the peer-to-peer network regardless of the implementation of the rest of this project. It is common in practice to create

library bundles in a generic form so that only one instance needs to be installed, and that its resources are shared between all the bundles that use it. While this is generally expected as part of software design, it should be noted that library bundles in OSGi can behave much like a daemon, supplying running services to application bundles, and consideration must be made for these services to be effectively shared.

3.1.3 JXTA R-OSGi Bridge

This bundle will carry the implementation of a `NetworkChannel` as discussed in section 3.1.1. Figure 3.4 illustrates the expected interface traversal after inserting the `JXTANetworkChannel`. This bundle is expected to be a very thin interface between the two, providing only the `NetworkChannel` implementation and any behavioural code to manage parts of the JXTA network that it cannot autonomously manage itself. This should restrict modification to the R-OSGi bundle as much as possible.

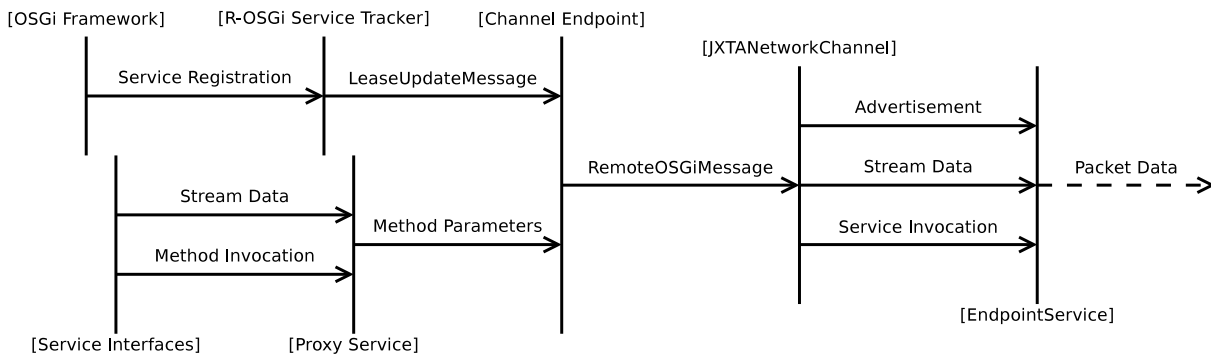


Figure 3.4: Interface traversal of R-OSGi with a JXTA NetworkChannel

3.2 Developing and Deploying OSGi bundles

The only requirement to turn a standard Java jar into an OSGi bundle is the addition of some metadata in the jar's manifest. These include the following directives:

- `Export-Package` which lists all the packages the bundle will supply to others within the framework.

- `Import-Package` lists the packages required by the bundle before it can be resolved.
- `Private-Package` lists packages included in the bundle that are not to be exported.
- `Bundle-Activator` names a class that implements the `BundleActivator` interface, that will be used to begin execution in the bundle.

Many other directives are defined in the OSGi specification [2] allowing for great flexibility when defining a bundles dependencies and behaviour.

The most significant difference between standard Java artefacts and OSGi bundles is how classes are supplied to a bundle's classpath. The classes contained within a bundle must either be listed in the `Export-Package` or `Private-Package` directive to be available to the bundle's classloader. This also includes any library jars included in the bundle. When a bundle is being resolved, it gathers other bundles that satisfy its dependencies as listed in the `Import-Package` directive. The bundle classloader delegates the loading of imported classes to the classloader of the bundle containing the imported packages.

Apache Maven¹ is a build tool for Java. Unlike other build tools, Maven uses a *project object model* to specify a mapping between source artefacts and the final build artefacts. This is in contrast with other tools that require the build process to be scripted. All maven artefacts are stored in repositories which are indexed and accessed on demand when a Maven project specifies a dependency.

The Maven Bundle Plugin² takes some of the work out of producing the required metadata for OSGi bundles. The plugin analyses the package imports from the Java source code and generates the appropriate manifest directives. This plugin also refits the repositories used by Maven with OSGi Bundle Repository (OBR) metadata. When the OBR client bundle is installed in an OSGi framework, bundles from a linked repository can be installed on demand much like an operating system's package management.

¹Maven - <http://maven.apache.org/>

²Maven Bundle Plugin - <http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html>

3.3 Including Native Code within an OSGi Bundle

The Java Native Interface (JNI) allows applications running inside a Java virtual machine to execute code native to the host platform. This provides a few advantages over pure Java in embedded devices including direct access to hardware and reduced overhead from the virtual machine.

The Linux Robotics Framework³ forms a hierarchical collection of components intended to be used for hobbyist robotics. This includes numerous device drivers as well as more abstract components building on the functionality of the lower level. This library will be the target to be included in an OSGi bundle.

Another plugin available for Maven will be used for this situation. The FreeHEP Native Archive⁴ plugin arranges the generation of native header files and the compilation of the native library. The output from this plugin can then be included in the artefact created by the bundle plugin.

When the Java virtual machine loads a native object, it cannot be hidden inside a jar. The operating system must be able to see the object file in order to link against it. This is a problem as OSGi bundles only ever consist of the one jar file. Native code is also tied to the particular architecture it was compiled for which severely impacts the portability of the bundle.

³Linux Robotics Framework - <http://www.linux-robots.com>

⁴FreeHEP NAR - <http://java.freehep.org/freehep-nar-plugin/intro.html>

Chapter 4

Results

4.1 Developing OSGi bundles

The main goal of investigation into development techniques was to produce a 'one click' build procedure which attempts to cover everything from compilation of source artefacts and deployment of the final artefacts in to a repository. It is possible to achieve this with Apache Ant, but this requires careful scripting of the entire process. This is a very tedious exercise and it is difficult to reproduce over multiple projects. Maven on the other hand only requires a single configuration file (*project object model*) which is easily modified to suit completely different bundles. Figure 4.1 shows a typical project object model for an OSGi bundle.

The Maven Bundle Plugin makes generating metadata very simple and includes publishing the final artefact to a repository. Combining this with the OSGi Bundle Repository installed inside a client framework, new application versions can be compiled, deployed and automatically installed on the clients with a single click. This is all accomplished by the XML in the `<plugin>` tags in figure 4.1.

Deployment of bundles can also be done over a peer-to-peer group, much like traditional P2P file-sharing. This topic has been investigated separately from R-OSGi - instead, concentrating on the sharing of bundle artefacts rather than services [10]. If these techniques can be implemented over an R-OSGi based network, this would lead to a truly peer-to-peer infrastructure with no need for a central bundle repository.

```

<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.osgibots</groupId>
  <artifactId>Robot</artifactId>
  <packaging>bundle</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Robot</name>
  <url>http://www.osgibots.com</url>
  <dependencies>
    <dependency>
      <groupId>org.osgi</groupId>
      <artifactId>org.osgi</artifactId>
      <version>3.0</version>
      <type>jar</type>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>>true</extensions>
        <configuration>
          <instructions>
            <Export-Package></Export-Package>
            <Private-Package>com.osgibots.*</Private-Package>
            <Bundle-Activator>
              com.osgibots.robot.Activator
            </Bundle-Activator>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>

```

Figure 4.1: An example project object model

4.2 Scalability Issues

When R-OSGi binds with a peer, all services on the server that have been exported are transferred to the client. This entails proxy bundles being generated for each bundle involved in a remote service. The developers have considered this and have worked into their service discovery mechanism, the ability to selectively bind to particular services, therefore only transferring proxies that are requested. Unfortunately, by attempting to replace this service discovery with my own JXTA implementation, this mechanism is lost. Figure 4.2 illustrates the situation when R-OSGi itself forms the only service advertisement to be consumed. In a large network, the resources of individual nodes can easily become overwhelmed by the number of proxy duplicates.

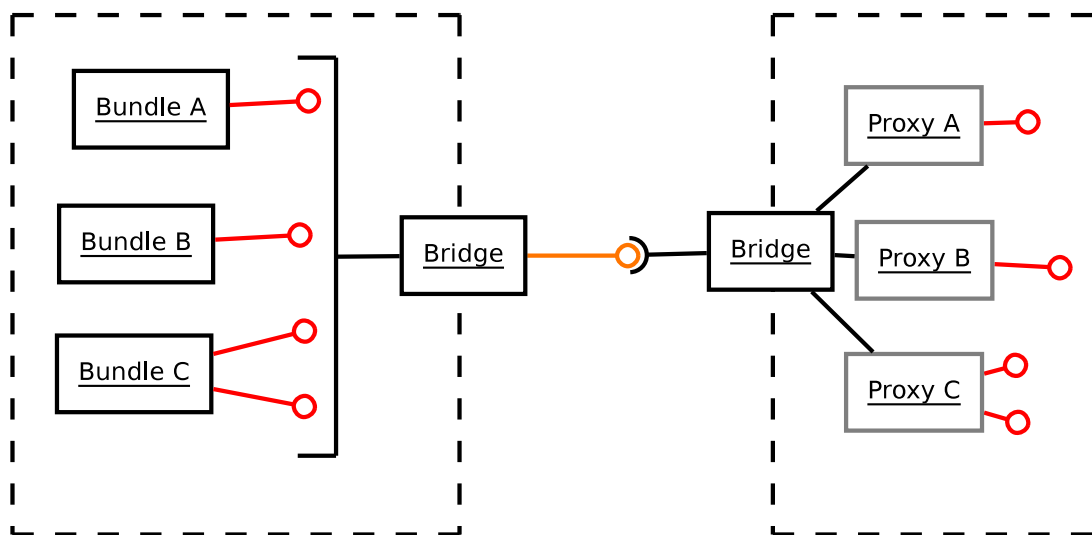


Figure 4.2: Exporting R-OSGi as a JXTA service

The alternative is to expose each service independently, with their own advertisement. Figure 4.3 illustrates this. With each service independently exposed, the service discovery mechanism can cherry pick the services of interest to a particular node. This will greatly reduce the network congestion from transferring proxy bundles, and will reduce the number of bundles installed in each node. This is discussed in more detail in section 5.1.

R-OSGi also uses persistent TCP sockets to reduce the overhead of binding the socket each time, increasing the performance overall. Unfortunately in a peer-to-peer ar-

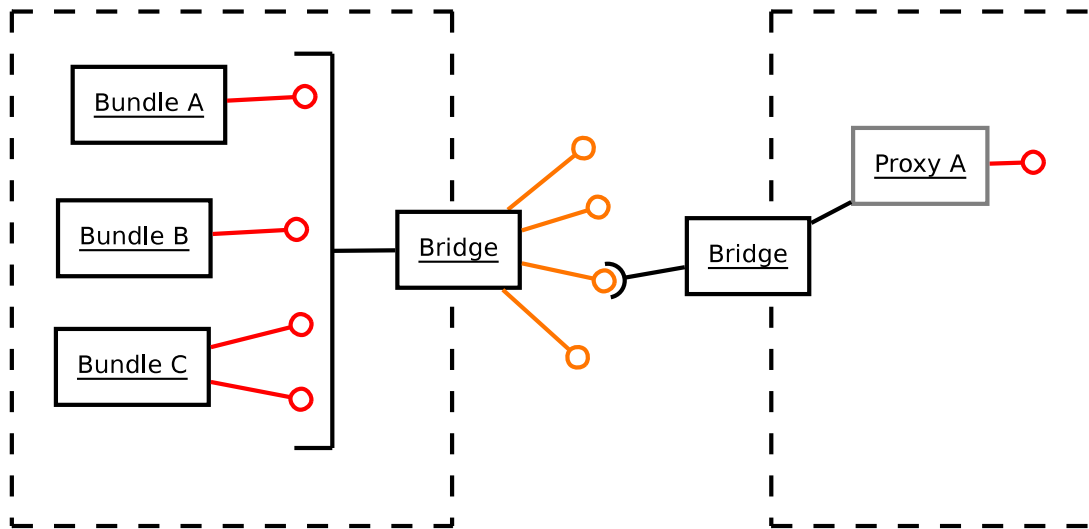


Figure 4.3: Mapping services to JXTA advertisements

management with inter-peer routing, the number of open communication lines quickly grows. It is to be assumed that nodes will attempt to minimise calls to remote services, and that the total time spent binding sockets is insignificant. For this reason, connections will only be kept open while they are being used. R-OSGi must still be able to communicate events such as the un-registering of a bundle. A specific advertisement for the R-OSGi daemon itself must be present on all nodes, in addition to one for each service, to allow these framework events to be communicated.

4.3 Preserving Language Semantics

OSGi is quite a change to the standard Java language. However it was expected that the underlying semantics would remain the same. There are a few things to look out for, especially when working with R-OSGi.

Because OSGi provides each bundle with its own classloader, developers must be cautious when using static objects. Static objects are expected to only be allocated once in an application, and all subsequent operations are performed on the same physical object. The actual situation is that a static object is allocated once *per classloader*. This means that multiple copies of the same static object may be created within the Java virtual machine.

Each classloader is expected to delegate loading imported packages to the bundle that exports them, which means that only one instance of each class is created when it is shared. However if two bundles both export the same package, then multiple instances of a static objects can exist.

Developers familiar with Java will take it for granted that when they pass an object as a parameter, it is in fact being passed by reference. They would expect that any modifications to the referenced object made during a method call would remain when the method returns. R-OSGi allows objects to be passed to remote services by serialising them and recreating them on the remote side. This is done transparently to the developer.

The remote service may then perform actions on the object, store it for later reference, or even expect to use it to perform a callback. Once the object is recreated on the remote side it has no synchronisation with its original copy. This is the result of distributed memory access issues as discussed in section 2.1.

4.4 OSGi with Native Libraries

The challenges faced with native code in an OSGi bundle were how native object files were accessed by the operating system from within the bundle, and how the bundle could remain portable when its native code was tied to a particular architecture.

The OSGi standard defines the `Bundle-NativeCode` directive which allows for some native object files to be listed along with the operating system and CPU architecture they are compiled for. The OSGi framework extracts these objects from the jar and writes them to a local cache where they are visible to the operating system. The framework then intercepts any calls to `System.loadLibrary()` and ensures the correct object file is loaded.

I was able to demonstrate this, maintaining a 'one click' build process and having the resulting bundle installed to an OSGi bundle repository. The bundle could be installed over a network and the native libraries were extracted by the OSGi framework. Services could then be implemented to expose the native drivers to other bundles including R-OSGi. Despite the bundle's dependency on a native object, the service

distribution does not require that object be duplicated as the remote calls are really remote procedure calls to the service, and the native call is made locally.

One limitation of this is that each framework instance requires a version of the native object for the architecture on which it is running. This requires a truly portable bundle to include the binaries for any architecture it is likely to be run on, possibly making it considerably larger. An alternative is to publish multiple versions of the bundle with different versions of the native objects. This requires more consideration for the infrastructure but it does make the bundle the clients receive much more manageable. Each version of the bundle can be labelled with the operating system and CPU architecture it is built for in order to be discernible. This distinction can be reflected in the OSGi bundle repository metadata, allowing deployment to be just as easy as any other bundle.

Chapter 5

Future Work

5.1 Mapping OSGi services to JXTA advertisements

Publishing R-OSGi directly as a JXTA resource leaves some issues un-addressed. The most concerning of these is the scalability of R-OSGi in a massively distributed environment. Without exposing the services available on each node, remote nodes cannot make an informed decision weather to bind to the R-OSGi service. This is likely resolved by binding to all available services and given the routing mechanism of JXTA, this means binding to every peer simultaneously. In addition, every service bundle from the remote peers will have a proxy constructed locally which will add to the demand on system resources.

The best solution to this is to make each exported service visible to initial discovery events. In JXTA this is in the service advertisements. In order to map each exported service to an JXTA advertisement, a more invasive modification to R-OSGi is required. The `NetworkChannel` interface does not provide enough information to manage these advertisements easily. The `ChannelEndpoint` interface (figure 5.1) sits between the various R-OSGi inputs and the network channel. It provides the most control over the communication channel and is likely the most appropriate place to insert a JXTA implementation. Figure 5.2 illustrates the interface traversal of R-OSGi with a `JXTAChannelEndpoint`.

The default implementations of the `ChannelEndpoint` and `NetworkChannel` come close

```
public interface ChannelEndpoint {
    void receivedMessage(RemoteOSGiMessage msg);
    Object invokeMethod(String service, String methodSignature,
        Object[] args) throws Throwable;
    Dictionary getProperties(String service);
    Dictionary getPresentationProperties(String service);
    void trackRegistration(String service, ServiceRegistration reg);
    void untrackRegistration(String service);
    URI getRemoteAddress();
    void dispose();
}
```

Figure 5.1: ChannelEndpoint interface definition

to two thousand lines of code, presenting no small task to replace. The `ChannelEndpoint` is also not designed to be easily replaced and would require significant modification to the R-OSGi bundle.

Service discovery using SLP has been developed for R-OSGi taking advantage of an LDAP style naming of services. This naming style allows an unambiguous mapping between SLP services and R-OSGi services [11]. While this doesn't include the various features JXTA provides, it does solve scalability issues with R-OSGi.

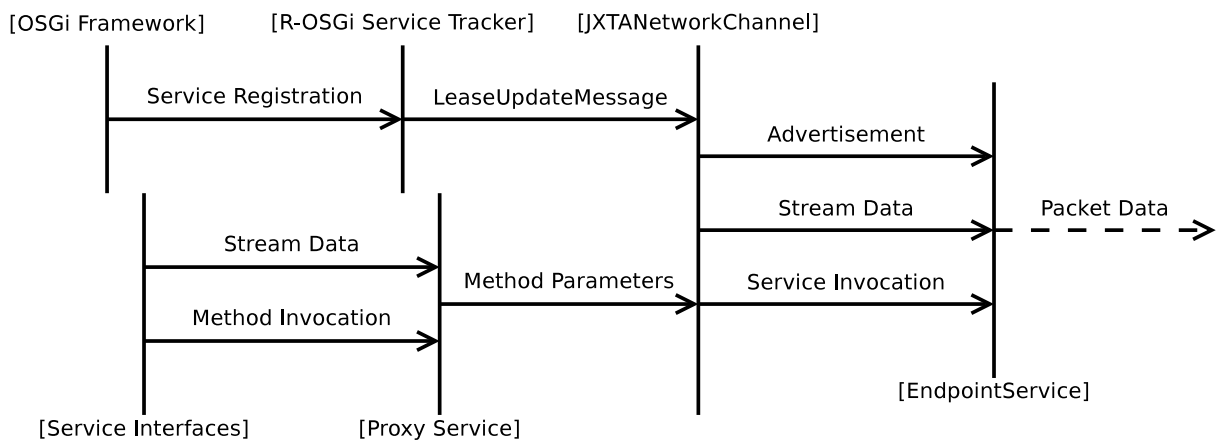


Figure 5.2: Interface Traversal of R-OSGi with a JXTA ChannelEndpoint

Chapter 6

Conclusion

Despite still suffering some of the issues with distributed computing discussed in section 2.1, R-OSGi and the OSGi framework form a very powerful distribution mechanism. Combined with very well thought out developer tools and deployment infrastructure, OSGi based applications are an absolute pleasure to work on.

R-OSGi attempts transparency in a distribution mechanism to the extent of the OSGi paradigm. Unfortunately the remote calls can still have unexpected side effects that local calls would not. Forced transparency in this case can be dangerous as the developers themselves are unaware of what to avoid. In keeping with the conclusion made by the authors of [1], local and remote objects cannot be treated the same.

OSGi provides great support for using native code within a bundle. The ability to package native code inside a bundle and publish it to a repository was demonstrated with various Maven plugins. The portability of the bundles can also be maintained and the services provided by a native are compatible with the R-OSGi network. This greatly increases the potential of R-OSGi on embedded devices.

The security mechanisms based on public key cryptography in both OSGi and JXTA provide a very robust platform for developing hierarchical authorisation for both code deployment and peer-to-peer communication.

A JXTA based communication layer may work, but will probably suffer slow performance when compared with an SLP based approach. The main motivation for using JXTA is convenience as it supports managing all levels of the peer-to-peer network.

It would be possible to implement all the features of JXTA above an R-OSGi/SLP network resulting in a 'specialised' protocol that may not be portable to other applications. Given that R-OSGi distributes arbitrary services anyway, this is not at all limiting.

Bibliography

- [1] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical report, Sun Microsystems Labs, 1994.
- [2] OSGi Alliance. *OSGi Service Platform Core Specification*, R4.1 edition, April 2007.
- [3] Jan S. Rellermeier, Gustavo Alonso, and Timothy Roscoe. R-OSGi: Distributed Applications through Software Modularization. In *ACM/IFIP/USENIX 8th International Middleware Conference*, 2007.
- [4] Chuan Zhu and Guiran Chang. A JXTA-based Communication Model for OSGi Framework with Extension to P2P Networks. In *ISECS International Colloquium on Computing, Communication, Control, and Management*, volume 1, pages 215–219, August 2008.
- [5] C. Loeser, R. Schaefer, and W. Müller. JXTA for Virtual Home Environments.
- [6] Jan S. Rellermeier, Michael Duller, and Gustavo Alonso. Engineering the Cloud from Software Modules. In *Workshop on Software Engineering Challenges in Cloud Computing (ICSE-Cloud, in conjunction with ICSE)*, 2009.
- [7] Brendon J. Wilson. *JXTA*. New Riders Publishing, June 2002.
- [8] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. 1996.
- [9] Sun Microsystems Inc. *JXTA v2.0 Protocols Specification*, v2.0 edition, October 2007.
- [10] Stéphane Frénot and Yvan Royon. Component deployment using peer-to-peer overlay. In *Working Conference on Component Deployment*, November 2005.

- [11] Jan S. Rellermeyer and Gustavo Alonso. Services Everywhere: OSGi in Distributed Environments. In *EclipseCon*, 2007.