R-OSGi: Distributed Applications Through Software Modularization

Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe

Department of Computer Science, ETH Zurich
8092 Zurich, Switzerland
{rellermeyer, alonso, troscoe}@inf.ethz.ch

Abstract. In this paper we take advantage of the concepts developed for centralized module management, such as dynamic loading and unloading of modules, and show how they can be used to support the development and deployment of distributed applications. We do so through R-OSGi, a distributed middleware platform that extends the centralized, industry-standard OSGi specification to support distributed module management. To the developer, R-OSGi looks like a conventional module management tool. However, at deployment time, R-OSGi can be used to turn the application into a distributed application by simply indicating where the different modules should be deployed. At run time, R-OSGi represents distributed failures as module insertion and withdrawal operations so that the logic to deal with failures is the same as that employed to deal with dependencies among software modules. In doing so, R-OSGi greatly simplifies the development of distributed applications with no performance cost. In the paper we describe R-OSGi and several use cases. We also show with extensive experiments that R-OSGi has a performance comparable or better than that of RMI or UPnP, both commonly used distribution mechanisms with far less functionality than R-OSGi.

1 Introduction

Modular design is a cornerstone of software engineering, and much effort has been invested in concepts and tools to manage modules and the dependencies among them. Nowadays, modularization pervades programming languages, development environments, and even system architectures. In particular, recent years have seen the emergence of "module management systems" which handle loading and unloading of modular program units at runtime, and dynamically creating and destroying bindings between services in different modules.

In this paper we explore using centralized module management as the basis for the design and deployment of distributed applications. Our work is based on the OSGi specification [1], a widely used module management API designed to work on a single system that we extend extend to work in a distributed setting.

R. Cerqueira and R.H. Campbell (Eds.): Middleware 2007, LNCS 4834, pp. 1–20, 2007.
© IFIP International Federation for Information Processing 2007

The key insight is that the module boundaries instituted by centralized module management systems are generally well-suited to being repurposed as distribution boundaries. In the past, networked applications have typically distributed their functionality by interposing communication proxies at procedure calls or object method invocation, with mixed results. In particular, the issue of transparency has dogged distributed computing platforms based on these models: as Waldo et. al. [2] point out, a remote procedure invocation has fundamentally different semantics to a local call, and consequently fundamentally different exception handling code must be written by the programmer.

In contrast, module management systems like OSGi are designed to handle unloading of modules at any time, and include event notification functionality to enable programmers (indeed, to require them) to cleanly handle services disappearing without notice. We take advantage of this by representing communication-related failures as local module unloading events.

By doing so, we effectively turn software modules into the potential units of distribution. The result is *Remoting-OSGi* (R-OSGi), a distributed middleware platform that can transparently distribute parts of an application by simply distributing its software modules. R-OSGi is a middleware layer on top of OSGi. This matches the lightweight design of OSGi and allows us to use R-OSGi on any OSGi enabled application.

R-OSGi makes the following contributions:

1. *Seamless embedding in OSGi*: From the OSGi framework's point of view, local and remote services are indistinguishable. Existing OSGi applications can be distributed using R-OSGi without modification.
2. *Reliability*: The distribution of services does not add new failure patterns to an OSGi application. Developers deal with network-related errors in the same way they deal with errors caused by module interaction.
3. *Generality*: The middleware is not tailored to a subset of potential services. Every valid OSGi service is potentially accessible by remote peers.
4. *Portability*: The middleware runs Java VM implementations for typical resource-constrained mobile devices, such as PDAs or smartphones. The resource requirements of R-OSGi are by design modest.
5. *Adaptivity*: R-OSGi does not impose role assignments (e.g., client or server). The relation between modules is generally symmetric and so is the distributed application generated by R-OSGi.
6. *Efficiency*: R-OSGi is fast, its performance is comparable to the (highly optimized) Java 5 RMI implementation, and is two orders of magnitude faster than UPnP.

In the next section we discuss in more detail the relevance of module management systems for distributed applications, using OSGi as a case study. In Section 4 we discuss the architecture and design of R-OSGi, and describe the implementation in detail in Section 5. Section 6 presents performance results for R-OSGi, and Section 7 details several use cases including ubiquitous computing

devices and a tool for refactoring a large, pre-existing OSGi-based application in Eclipse to run in a distributed setup. We conclude in Section 8.

2 Background

Models and frameworks for building distributed systems have a long history. The conventional approach is to make remote invocations identical to local procedure or method calls, as exemplified by Remote Procedure Calls (RPC) [3], Java Remote Method Invocation (RMI), the Common Object Request Broker Architecture (CORBA) [4], or the Distributed Component Object Model (DCOM) [5]. While providing a form of distribution transparency at the level of invocations, the application must nevertheless be manually factored into distributed components, and the large-scale structure of the application usually reflects this factoring. The same is generally true for analogous operating system-based approaches, such as Amoeba [6] or SOS [7].

Alternatively, centralized applications written in a component framework can be automatically factored into distributed components. Coign [8] partitions COM-based Windows applications into two parts that can be distributed in a client/server configuration. Coign instruments the code through binary rewriting, analyzes the dependencies between COM components and calculates a graph-cutting according to a cost metric for introducing network communication between the subgraphs. Similarly, JOrchestra [9] automatically partitions a program by rewriting bytecode to replace local methods with remote invocations, and object references with proxy references. In these approaches, the distribution is orthogonal to the original design, and occurs along object boundaries which were typically not designed with distribution in mind, giving rise to the kind of transparency and performance problems described in [2].

Recent centralized module management systems, e.g., MJ [10] and OSGi, in contrast to typical component frameworks, impose boundaries between modules which are explicit at the level of program code. This is done to better deal with dynamically loading, updating, and unloading of modules at runtime. We describe OSGi in more detail below, as it forms the basis of our system.

However, we note that to date, efforts to add distribution support to OSGi have either followed the OSGi specifications in providing protocol adapters to existing Jini [11] and Universal Plug and Play (UPnP) [12] infrastructures, or (as with the Newton Project [13]) introduce an additional component model for distribution independent of OSGi's module boundaries and based on an existing infrastructure like Jini. Both approaches are what might be termed "invasive": they require the application to be explicitly structured (or restructured) around the distribution model provided by Jini or UPnP, and hence the application must be factored in such a way as to conform to one of these component models. What is clearly missing is a way to have across remote OSGi instances without loosing the generality of the OSGi model, or, equivalently, to allow an OSGi application to be easily distributed along OSGi module boundaries. Filling this gap is the main result of this paper.

3 Overview of OSGi

Before discussing the design and implementation of R-OSGi, we briefly describe all the relevant aspects of the OSGi model. The OSGi specification is maintained by the OSGi Alliance (including vendors and users). OSGi is used in a number of systems (e.g., Eclipse [14]) and several open-source implementations exist, such as Apache Felix [15], Knopflerfish [16], and Concierge [17].

3.1 Basics of OSGi

OSGi is both (1) a programming model to develop Java applications from modular units (*bundles*) decoupled through *service interfaces*, and (2) a runtime infrastructure or *framework* for controlling the life cycle of bundles. Among other features, OSGi allows developers to dynamically manipulate bundles: new bundles can be added and existing bundles updated or removed all at runtime. OSGi maintains consistency across modules by keeping track of the dependencies between modules.

As in systems such as Tomcat [18], OSGi implements module management by using a separate class loader per bundle and disposing of the entire class loader when the bundle is unloaded. However, unlike Tomcat, where shared code has to be placed into the scope of a special *shared* class loader, all bundles loaded by an OSGi framework are allowed to define shared Java packages and interact through services.

3.2 OSGi Services

OSGi implements a centralized service-oriented architecture with loosely coupled *services* (Figure 1). In the OSGi model, any Java class can be published as a service to be used by other bundles in the system. Typically, a service includes an implementation (an instance of a class), one or more service interfaces under which the service is published, and a set of service properties. The OSGi framework maintains a *registry* of all services published in the system. Bundles can retrieve services by the name of their interface, and optionally use LDAP-style RFC 1960 [19] filter predicates on service properties for higher selectivity.

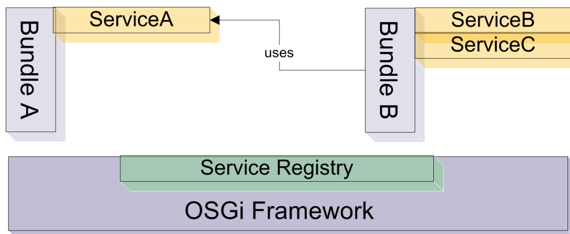

Fig. 1. OSGi Framework with Bundles and Services

Over the indirection of a service reference, a client bundle can bind to the service object and invoke operations on it from its own code. Since services might not be present or even disappear during the life cycle of the service's client, access to service objects must be mediated and controlled. OSGi does this by sending events whenever the state of a service changes. The typical pattern of service usage is to listen to such events and either disable parts of the requesting bundle when the requested service becomes unavailable or even trigger a halt of the whole bundle, if the presence of the service is required for correct operation of the bundle.

3.3 The OSGi Whiteboard Pattern

Besides services invoked from other bundles, OSGi services can also be used to simplify different variations of producer/consumer exchanges. Typically, the publish/subscribe pattern is used for this purpose: each event source maintains its own registry of subscribed listeners and delivers events to all subscribers as the events take place. The whiteboard pattern [20] used in OSGi simplifies this process. Instead of requiring each listener to subscribe to individual events and the source to hold the subscriptions, the OSGi *service registry* is used. Listeners register themselves under a specific listener service interface. Once this is done, the listener is not required to dynamically track all sources of events, instead, it has implicitly acquired a global subscription to all existing and future event sources. The OSGi registry is thus the whiteboard to which all listeners may subscribe. Event sources can retrieve all registered listeners whenever an event occurs. With such an approach, the coupling between listener and source is reduced to a minimum and the listener can place the subscription even if no source is currently present. It has been shown that the whiteboard pattern is often more efficient than traditional publish/subscribe in terms of code size and the total resulting number of classes [20].

4 The R-OSGi Approach

R-OSGi allows a centralized OSGi application to be transparently distributed at service boundaries by using proxies. Figure 2 shows a simplified example with one service provider (I) and one service consumer (J). To bundles on peer J, the R-OSGi proxy is indistinguishable from local OSGi services such as service A and B. The R-OSGi protocol on the proxy is used to make remote invocations to the original service, which is located on peer I, and events from I are transparently forwarded to J and occur as if they were issued by a local bundle. The only difference between local and remote services are additional properties that allow services aware of distribution to perform specialized operations, e.g., for system management.

R-OSGi uses four principal techniques to achieve the goal of transparency: (1) *dynamic proxy generation* at bind-time for cross-network invocation of services, (2) a *distributed Service Registry* based on SLP complementary to the

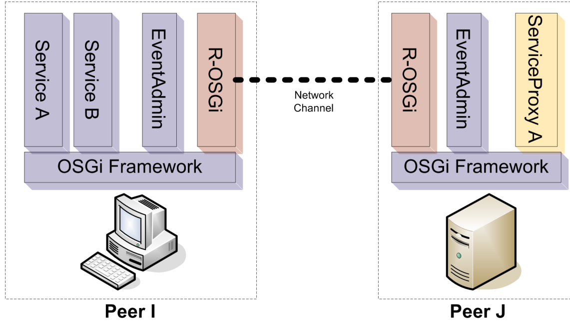

Fig. 2. Architectural Overview

centralized version in OSGi, (3) *mapping network and remote failures* to local module hotplug events, and (4) *type injection* to resolve distributed type system dependencies. We describe these in turn.

4.1 Dynamic Service Proxies

R-OSGi creates transparent client proxies for remote services on the fly. To a service client, these proxies behave as a local service and are also provided by locally-instantiated bundles. However, a proxy bundle redirects all service method calls to the original service residing on the remote machine and propagates the result of the method call back to the local client.

The approach of dynamically generating the proxy code at the client facilitates spontaneous interaction between services, but also reduces to a minimum the data (in the form of Java bytecode) that must be stored on the server or transferred over the network when a client binds to (or *fetches*) a service.

The typical information required to create a proxy for a particular service interface is determined by bytecode analysis of the original service when it is registered. When a client fetches the service interface, the service provider responds with the corresponding Java bytecode for the interface along with any serialized properties of the service.

From the interface bytecode, the client can then generate a full proxy for the service. No precompiled skeletons or stubs need to be provided by the implementor of the service, and no actual proxy code needs to be transferred. This is particularly useful in the case of *servers* running on resource-constrained devices, since the service provider bundle does not need to retain any code for the client proxy.

4.2 Service Registration and Location

OSGi is built around a centralized service registry. In order to transparently distribute OSGi applications, a distributed registry implementation is required. It is not possible to make a distributed service registry look like a local registry

without changing the OSGi framework implementation. Thus, to avoid limiting the generality of the platform, R-OSGi works with a complementary service discovery protocol and builds proxies for remote services which then register their services with the conventional OSGi service registry. Hence, conventional OSGi bundles can be used (and distributed) in R-OSGi without modification.

OSGi uses an explicit binding model whereby the client bundle invokes (as a synchronous method call) the service registry, which hands over a set of *service references* in return. The request contains two arguments: the class name of the requested service and a filter expression which can, for instance, be used to distinguish between equivalent implementations of the same service type. Filters are based on the LDAP filter syntax (RFC 1960 [19]). A client in possession of a valid service reference can then attempt to establish a binding to the service, and afterwards invoke operations on it.

While the explicit binding model simplifies the handling of network and remote node failures in R-OSGi (see Section 4.3), the approach of building proxies for services introduces a potential scalability problem since in a large distributed system there might be a large number of nodes, and a large number of services. Each service proactively announcing its availability and the system generating proxies for every available service might increase network traffic, and tie up processing resources at the nodes.

R-OSGi's distributed service registry alleviates this problem by making service discovery (and thus the proxy generation) reactive. Bundles can register services of type `DiscoveryListener` and set properties to convey information about the service interfaces they are interested in, optionally including a filter string. Following the whiteboard pattern, R-OSGi keeps track of all registered listeners by observing service registration events from the local service registry. It initiates remote service discovery whenever there is an entity in the system that has announced a demand for a service.

Likewise, peers announce their offers of services to the network and allow remote access to them according to a locally-determined policy. Whenever a new service is registered with the local framework with properties that indicate it should be offered remotely, R-OSGi triggers registration of this service with the remote service discovery layer.

Explicit determination of which services to offer for remote access in this way can be performed by the application, at the cost of loss of transparency (since the application must set the required properties). Alternatively, a surrogate bundle separate from the application but residing on the same node can listen for local service registrations, and selectively re-export some services remotely without requiring the application itself to be distribution-aware.

4.3 Transparent Distribution

Transparently distributing programs designed for a single address space context has been a problematic concept. Waldo et. al. [2] provide a good summary of the main problems: networked systems are fundamentally different in behavior to centralized ones, and the semantics of an invocation are also fundamentally

different. Consequently, the argument goes, it is unlikely that a centralized program will perform with acceptable performance, let alone correctly, when factored into distributed components. The basic problems here are communication latency and non-determinism, and unreliability (either due to message loss or partial failure of the nodes or network). These arguments are powerful and persuasive.

R-OSGi sidesteps these issues by intelligently exploiting the way that OSGi programs are already written – the assumption of unknown performance characteristics of cross-bundle calls. Furthermore, rather than masking distributed failures, R-OSGi exposes these events to application bundles, but in a form that the bundle is already designed to handle: the disappearance of service bundles through module unloading.

R-OSGi conceptually maps failures arising from the distribution of components to local hot-plug events. From the OSGi model, developers are used to guarding the code against the case that parts of the system are not available. Usually, this is done by listening to service events or using the OSGi *ServiceTracker*. In a purely local configuration, services can become unavailable when some entity in the system, in particular a user of the system, decides to stop or to uninstall the bundle that has provided the service. By mapping network malfunctions to these events that are already handled by the applications, we introduce no failure patterns that are not already possible in purely centralized situations.

For instance, if a service providing peer fails, we detect the breakdown of the network channel and the failure to reconnect. Having observed this, R-OSGi immediately uninstalls the proxy bundle. Even if the network operates without failures, the original service can throw exceptions. We serialize these exceptions and rethrow them in the proxy bundle to mime the exact behavior of the original service.

OSGi Services give no guarantee about execution time regardless of whether they are local or remote. Side effects of services such as threaded design or database accesses (e.g., a persistence service), can lead to an execution time that appears to be non-deterministic from the client point of view. A user might even decide to replace a fast implementation of a service by an extended but overall slower implementation and the client has to live with this situation. Furthermore, services are often event-driven and since events in OSGi are typically dispatched asynchronously, no assumptions about timing can be made. This is a considerable difference to plain objects, that are most often expected to execute methods within a very short time. A further difference between R-OSGi in comparison to systems like CORBA is that the granularity of distributed entities is much larger. In OSGi, services encapsulate whole functional units and the dependencies between services are typically restricted to semantical dependencies at the application level. Objects in contrast tend to have a larger number and often nested interconnections that make bad effects of the network more severe.

4.4 Type Injection

In OSGi, all code is modularized into bundles and imports of code from other bundles have to be explicitly declared in the bundle JAR manifest. Several implications arise in the context of proxy bundles. The service interface might use types in method parameters or return values that do not belong to the standard Java classes and cannot be assumed to be present at the client. This can either be the case if the type is declared by a class of the original service bundle, or because the package to where the class belongs was imported by the original service bundle. It has to be assured that the generated proxy is resolvable, i.e., it contains all the types that are used by methods of the service. R-OSGi thus has a special strategy to ensure type consistency for the service interface. Type injection is used to make service proxies self-contained.

When the (remotely accessible) service is registered, every type occurring in the service interface is observed by a static code analysis. If the type is contained in the service bundle and the package is declared to be exported by the service bundle, the corresponding class is added to the so-called *injection list*. Referenced types not contained in the service bundle are left out. In a second step, the transitive closure of all injections is formed, once again distinguishing between own and imported classes. The injections are saved with the service registration. Whenever a client fetches the service, the injections are transmitted in addition to the service interface and the service properties. During proxy generation, the injections are materialized and stored in the proxy bundle. The packages of all referenced classes not included in the injections are declared as imports of the proxy bundle. The packages of all injected classes are declared as exports to ensure type consistency within the framework. Classes from the packages *java.** and *org.osgi.** are excluded from the whole process since it is assumed that they belong to the execution environment. The result of the injection strategy is a minimal set of classes and package imports that make the service proxy self-contained and resolvable.

Beyond the described code analysis to determine the minimal set of injections, service registrations can be manually provided with classes that have to be injected into the bundle. This can be useful in particular cases, e.g., if an argument of a service method is an interface and the service provider wants to add an instantiable implementation of this interface.

5 Implementation

5.1 Distributed Service Registry

R-OSGi implements the distributed registry using the Service Location Protocol (SLP) [21,22,23] as the underlying mechanism. We discuss the choice of SLP over more apparently natural choices like Jini in this section. Rather than using a C-based daemon implementation of SLP like OpenSLP, we instead developed a pure Java implementation, jSLP [24]. jSLP implements all the mandatory features of the SLP protocol, plus most of the optional features, yet has a code footprint of

only 55kBytes. We do not discuss jSLP further here for reasons of space; further information and complete source code can be found at [24].

SLP has several compelling features for R-OSGi: its adaptivity, the inherently distributed lookup process, and the similarity with OSGi in the naming of services. To use SLP as a fully decentralized service registry, we exploit the adaptive behavior of the SLP protocol. In SLP, when a dedicated *Directory Agent* is present, clients communicate exclusively with this central registry server. If no DA is present, the clients use multicast (as in SSDP [25]). Through this feature, R-OSGi implements a distributed SLP layer that can be used in a wide range of situations. In terms of naming, both OSGi and SLP identify a service by a single string. In OSGi, this is the fully qualified name of the interface under which the service has been registered. In SLP, the name is a service URL of the form `service:`*`serviceType`*`://`*`URL`* where the service type is of the form *`abstractType`*`:`*`concreteType`* By describing all OSGi services by the same abstract type `service:osgi` and using the fully qualified name of the interface as the concrete type, we have a bidirectional mapping between OSGi and SLP services. OSGi supports LDAPv2 filter predicates on service properties to allow more declarative and fine-granular services matching. This feature becomes particularly useful when the service registry is no longer a central but a large distributed one. With the choice of the SLP protocol that also supports LDAP filters over service attributes, R-OSGi leverages the power of expressive service predicate matching for the distributed case.

After a service is discovered, R-OSGi introduces an intermediate step before the actual service is delivered (i.e., imported into the local framework). This is important for security reasons as it allows users to, e.g., see the available remote services in a GUI before connecting to them. With such a step, R-OSGi matches the behavior of OSGi, which also uses an indirection over service references. R-OSGi also supports explicit connection to a remote peer if the application has *a priori* knowledge of the distribution of services in the system.

5.2 Network Channels and Message Transport

The communication structure of R-OSGi is purely message-based. For efficiency of parsing and handling, all messages are binary. Messages consist of a header that indicates the type of the message plus some common attributes, and a body with the parts specific to the message type.

Network channels in R-OSGi are by default persistent TCP connections using the TCP *keep-alive* option. As long as there is traffic within the timeout period, the connection is kept open. This reduces the overhead for the TCP handshake that would otherwise precede every call to a service. R-OSGi is nevertheless extensible. We have, for instance, implemented tunneling of R-OSGi messages through HTTP to support communication through firewalls.

When a connection through a network channel is established, the two peers exchange *symmetric leases* (Figure 3). A lease contains the names of the services that the peer offers as well as the event topics the peer is interested in. The latter is used in the context of remote events as discussed in Section 5.5. In R-OSGi,

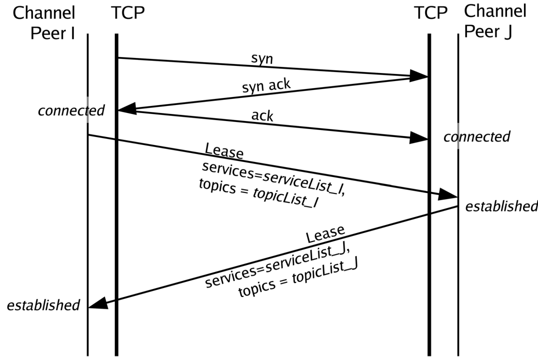

Fig. 3. R-OSGi Channel Establishment and symmetric leases

unlike in systems like Jini, a lease is more a contract between the two peers than a temporal limitation. Whenever changes to services or to subscriptions are announced through the lease, the peer that has issued the lease is obliged to invalidate the existing lease.

5.3 Proxy Generation

On the client side, the proxy is created through a *Proxy Generator*. The Proxy Generator is based on the *ASM* library [26], and it uses bytecode manipulation to create the service interface. First, an empty class is created that implements both the service interface as well as the OSGi specific interface (the *BundleActivator*). The OSGi specific parts, including the registration of the service with the local framework and retrieving the R-OSGi service, are implemented by emitting generic templates. Subsequently, every method of the service interface is visited and the corresponding method implementation created. Each method implementation delegates the method call to the network channel provided by R-OSGi and invokes the following method:

```
Object invokeMethod(final String serviceURL,
    final String methodSignature, final Object[] args)
    throws RemoteOSGiException;
```

The *serviceURL* is known at proxy generation time and hard-coded into the proxy, since every remote service gets its own proxy. The method signature is also a constant of each method implementation. The *args* array is built at runtime by aggregating the actual arguments.

The proxy-implementation of the service interface is packed into a JAR file together with the service interface. The required metadata is added to the manifest to turn the JAR file into a valid bundle. The service interface is then exported. This allows other bundles to import the interface if it is not yet known. Otherwise, the import statement is used and the newly created bundle is linked

against the existing interface to preserve consistency within the framework. R-OSGi stores the generated bundle and installs it, which leads to a registration of the proxied service. Since the service is registered under the transmitted interface name, local bundles cannot distinguish between a proxied service and a local service, thereby preserving full location transparency.

5.4 Method Invocation

Every method invocation corresponding to a remote service is transformed into the *invokeMethod* call shown above and sent through the underlying R-OSGi channel. On the other side of the channel, the first step taken is to lookup the corresponding service. R-OSGi holds references to all services that are released for remote access in a HashMap to guarantee a quick lookup. On this service object, a reflective call of the original method is performed. However, the Java reflection API requires the formal method parameters for matching and these can differ from the types of the actual arguments. This is particularly true if one of the formal parameters is an interface or an abstract class. One option would be that for every method call, the whole type hierarchy of each of the arguments is used for matching. To avoid this overhead, the signature of the method is part of the transmitted message. R-OSGi uses the signature to unambiguously match the original method. If the reflective method call succeeds, the result value is packed into a response message and sent back. If an exception occurs, the exception object is serialized, packed into the response message, and thrown on the other side of the channel. This makes the syntactic behavior of the remote service indistinguishable from that of local services.

5.5 Remote Events

As in UPnP, R-OSGi implements both remote service invocation as well as an event based architecture. R-OSGi uses the OSGi concept of events as described in the R4 specification of the *EventAdmin* service. In R-OSGi, the EventAdmin service is implemented as a whiteboard pattern over the distributed service registry. A bundle registers for an event by registering an *EventHandler* service together with the property *event.topics* and the optional property *event.filter* stating a filter that is matched against the property set of occurring events. Topic strings of events follow a hierarchical structure and can be matched using wildcards. Bundles initially register the EventHandler in the local service registry. The subscription is announced to peers through a symmetric lease transmitted during the connection phase. On the other side of the channel, an EventHandler is registered locally for the stated topics and if any matching events are outstanding, they are sent back through the channel. To publish an event, bundles post it to the local *EventAdmin* service which then sends it to all registered listeners.

(a) connecting to service (b) displaying a robot service

Fig. 4. ServiceUI for an R-OSGi-driven Lego Mindstorms Robot on a Zaurus PDA

5.6 Presentations

The fact that R-OSGi modules are treated as units for distribution offers unique opportunities to specialize some of these modules. One such specialization in R-OSGi is the idea of *presentations*. A presentation is a single class with an associated user interface that can be downloaded by the client rather than simply used remotely. Services can attach presentations by setting the property *service.user_interface* to the fully qualified name of a class implementing the interface *ServiceUIComponent*. Declared presentations are automatically injected into the proxy bundle and registered as services in a whiteboard fashion. On the client side, it is possible to run the optional R-OSGi *ServiceUI* bundle. This bundle displays the information about discovered services and allows the user to fetch these services. If the service has a presentation attached, the graphical component provides a Java AWT panel. This panel is displayed in a tabbed environment to allow the user to interact with multiple remote services.

We have used presentations for controlling smart devices. Figure 4 shows the screen of a PDA that connects to a Lego Mindstorms robot through R-OSGi. The software controlling the robot is developed using R-OSGi and contains a presentation with the user interface to control the robot. The PDA first connects to the *Robot Service* as a normal R-OSGi service (Figure 4.a). It then downloads the presentation with the robot controller interface which now runs locally and allows the PDA to become the robot controller. As the example shows, with R-OSGi presentations and the ServiceUI, it is possible to implement the idea of the universal remote control that can connect to any kind of (R-OSGi enabled) smart device and control it. The user interface for a service comes directly from the device and thus allows to connect to previously unknown devices without any need for configuration or installation of device drivers. Note as well that the demands on the developer are very small as it is only necessary that the user

interface is designed as a module, something that it is likely to happen regardless of whether R-OSGi is used.

6 Experimental Evaluation

Since there is no standard benchmark for evaluating the performance of R-OSGi, we have adapted two suitable benchmarks from other areas.

The Javaparty/KaRMI [27] benchmark measures the performance of alternative RMI implementations. It calls various methods of a sample object using arguments of different size and complexity. We have implemented the Javaparty benchmark as an OSGi service which is transparently distributed by R-OSGi. For comparison, we have also implemented it as a service object which is distributed by RMI and as a UPnP service accessible through the Domoware UPnP service implementation [28] for OSGi. The benchmark client calls the different methods multiple times and determines the average invocation time from the accumulated runtime. Most of the arguments are instances of primitive types or primitive arrays with increasing length. We skipped the parts of the performance benchmarks that are specific to the KaRMI system and not relevant to R-OSGi.

The *WSTest* benchmark [29] measures the performance of web services. It was originally used to compare web service performance in Java and in .NET. The benchmark starts a number of agents that concurrently call one of four sample methods according to a predefined mix. The arguments of the method calls are complex objects. In the variant originally used by Sun and Microsoft, only one of the methods is called at once, concurrently by eight agents. Since UPnP is not able to use complex objects in service calls, we run this benchmark only for R-OSGi and RMI.

The benchmarks have been measured with the services running on an IBM R32 notebook with an 1.6 GHz Intel Pentium 4 Mobile CPU and with 512 MB RAM. The client was a Pentium 4, 3 GHz Desktop machine with 1 GB RAM. For the PDA tests a Sharp Zaurus 5500 with a StrongArm SA-1110 CPU running at 206 MHz and with 64 MB RAM has been used. In the notebook and the workstation, we use Sun J2SE 1.5 as the underlying VM. The Zaurus runs *cvm* [30], Sun's implementation of the CDC Personal Profile.

6.1 Service Binding

In a first experiment, we measured the *binding time*. In R-OSGi, this is the time spent to establish the connection, requesting the service, receiving the interface, and building the proxy. For RMI, this is the time needed to establish the connection and to download the stub from the codebase. The results are presented in Table 1. As the Table shows, R-OSGi performs better than RMI even though the client has more work to do. From our observations, the download of the stub is the source of the overhead in RMI. The differences between the two benchmarks are because the binding time depends on the complexity of the service and, in this case, this complexity is related to the number of service methods.

Table 1. Binding Time

Service	# Methods	**Binding Time in μs**	
		R-OSGi	**RMI**
JavaParty	7	147381	163702
WSTest	4	97147	168034

The benchmarks show that R-OSGi is more efficient in terms of binding time than RMI, an interesting result given the additional functionality that R-OSGi provides.

We have also tested how R-OSGi scales down to mobile devices by measuring the binding time for the Javaparty service on a Sharp Zaurus 5500 PDA with 802.11b wireless LAN. The measured binding time was 1585 milliseconds. This is a much higher overhead but comparable with the latency of such operations on mobile devices. Furthermore, this penalty has to be paid only once for each service.

6.2 Service Invocation

In a second experiment we compare the cost of invoking a remote service in R-OSGi, RMI, and UPnP using the Javaparty benchmark. Since UPnP does not support complex objects as arguments in service method calls, not all test methods of the benchmarks could be implemented for UPnP. The results are shown in Table 3. As the table shows, R-OSGi performs slightly better than RMI in many cases, especially when the arguments are complex objects. We also measured the round trip time in the test network which was 193 μs (+-7 μs). Compared with this value, the `ping()` method using R-OSGi has an overhead of only 1.5% whereas for RMI it is about 16%. Those tests that can be run with UPnP have an execution time two orders of magnitude larger than R-OSGi and RMI. The main reason is the high verbosity (resulting in higher network delays) and the expensive parsing of the XML involved.

A similar comparison was done using the WSTest benchmark (Table 2) where we measured both response time and throughput. R-OSGi has a lower response time per method and a higher throughput. We also tested the scalability of R-OSGi using this benchmark. The proposed setup of the WSTest specifications uses only eight agents. When, for instance, the echoVoid method is called by 80

Table 2. WSTest Benchmark Results

Test	**R-OSGi**		**RMI**	
	Resp.time (μs)	**Throughput**	**Resp.time (μs)**	**Throughput**
echoVoid	5799.109	1378.583	10914.879	732.583
echoStruct	11464.700	697.633	14067.500	568.533
echoList	12238.550	653.500	15390.130	519.767
echoSynthetic	2439.700	3275.567	3069.710	2604.667

Table 3. Javaparty Benchmark Results

Method invoked	Invocation Time in μs and STD		
	R-OSGi	RMI	UPnP
void ping()	195.813 ±0.52	225.18 ±0.738	87938.454 ±174.044
int ping()	214.633 ±0.479	227.98 ±0.645	87335 ±27.839
void ping(int)	216.838 ±0.43	227.172 ±0.789	87844.286 ±191.748
void ping(int, int)	227.043 ±0.427	228.885 ±0.509	88558.571 ±126.765
void ping(null)	202.974 ±0.393	228.031 ±0.472	-
void ping(Integer)	218.301 ±0.419	324.855 ±1.162	-
void ping(byte[1])	246.263 ±0.559	273.345 ±1.317	88770 ±122.241
void ping(byte[2])	246.237 ±0.425	273.656 ±0.54	88822.857 ±48.613
void ping(byte[4])	246.58 ±0.517	274.167 ±0.55	88832.857 ±40.958
void ping(byte[8])	247.94 ±0.51	274.41 ±0.514	88948.571 ±86.426
void ping(byte[16])	249.463 ±0.492	275.374 ±0.568	89088.571 ±39.071
void ping(byte[32])	252.988 ±0.514	277.174 ±0.681	89122.857 ±15.779
void ping(byte[64])	257.396 ±0.47	284.274 ±0.457	89055.714 ±19.166
void ping(byte[128])	270.142 ±0.704	295.539 ±0.591	89090 ±40.708
void ping(byte[256])	278.694 ±0.638	317.382 ±0.476	89162.857 ±38.439
void ping(byte[512])	337.612 ±0.818	363.596 ±0.692	89201.429 ±104.53
void ping(byte[1024])	429.258 ±0.966	457.977 ±0.947	89467.143 ±32.388
void ping(byte[2048])	532.447 ±1.031	582.424 ±1.19	89997.5 ±24.875
void ping(byte[4096])	692.89 ±1.072	718.177 ±1.158	91098.75 ±63.134
void ping(byte[8192])	1275.493 ±7.605	1095.5 ±2.291	98631.429 ±3185.514
void ping(byte[16384])	1903.204 ±11.198	1872.352 ±7.369	97718.571 ±36.027
void ping(byte[32768])	3941.772 ±65.534	3932.065 ±52.933	157588.571 ±93.263
void ping(float[1])	251.204 ±0.593	275.155 ±0.757	-
void ping(float[2])	252.204 ±0.574	276.011 ±0.5	-
void ping(float[4])	253.924 ±0.648	277.676 ±0.374	-
void ping(float[8])	256.831 ±0.526	279.994 ±0.796	-
void ping(float[16])	262.098 ±0.488	287.206 ±0.602	-
void ping(float[32])	273.858 ±0.5	297.677 ±0.662	-
void ping(float[64])	296.173 ±0.741	317.408 ±0.567	-
void ping(float[128])	344.244 ±0.701	369.27 ±2.416	-
void ping(float[256])	439.993 ±0.92	470.157 ±7.577	-
void ping(float[512])	551.247 ±1.21	605.09 ±9.467	-
void ping(float[1024])	723.892 ±1.592	749.488 ±3.622	-
void ping(float[2048])	1224.912 ±2.27	1251.543 ±10.059	-
void ping(float[4096])	1954.012 ±11.076	1945.257 ±38.723	-
void ping(float[8192])	4105.288 ±77.579	3982.534 ±59.839	-
void ping(float[16384])	8036.289 ±132.496	7916.875 ±132.722	-
void ping(float[32768])	13460.103 ±131.231	13839.062 ±104.921	-
void ping(DM(1024,1024))	918597.938 ±13063	923121.212 ±12276	-
void ping(DM(2048,2048))	3557125 ±16284	3614843.75 ±23682	-

concurrent agents, the response time for R-OSGi increases by only 5% whereas it increases by about 23% for RMI. This indicates that R-OSGi scales very well, even for large setups with massive distribution.

7 Use Cases

In this section we briefly present two use cases implemented with R-OSGi to illustrate its potential to distribute complex applications.

7.1 R-OSGi Deployment Tool

The first use case is a tool to help developers to distribute an application by *dragging and dropping* between a visualization of the modules of the application

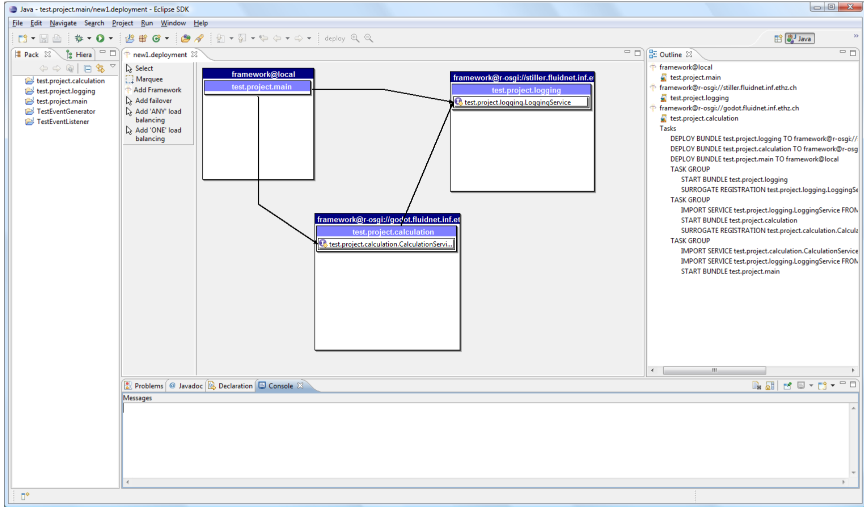

Fig. 5. Screenshot of the R-OSGi Deployment Tool

and a representation of the distributed nodes available. The tool has been written as an Eclipse *plugin* and provides an overview of all nodes running R-OSGi and where the application could be distributed. Through the R-OSGi capabilities to make explicit connections, the developer can add new OSGi nodes that are outside of the scope of service discovery. To distribute an application, the tool takes an ordinary OSGi application as input. It first analyzes the services and dependencies between the bundles. Then it graphically displays the architecture of the application as it would run on a single machine. The developer can then drag and drop bundles into the different available nodes. An example deployment with three bundles on two different nodes is shown in Figure 5. The tool visualizes all dependencies arising from this setup and gives the user an idea how many network communication is involved in a particular setup. Once the user commits a configuration, R-OSGi does all the work of deploying the bundles to the corresponding machines and creating surrogate registrations and discovery listeners. The result is the original application running in a distributed environment without requiring the developer to change a single line of code. The developer has full control on how the application is distributed. The tool is only intended to create static deployments. In the future, we will extend this tool to seamlessly introduce module replication, and allow the end-user to profit from fault-tolerance or load balancing by taking advantage of the distributed setup and the loose coupling of components.

7.2 R-OSGi Tea Machine

As an example of how to use R-OSGi with small devices, we have implemented a remote-controlled tea machine (along the lines of the Trojan Room coffee

(a) Tea Machine (b) Connected PDA (c) Presentation

Fig. 6. R-OSGi-driven Smart Tea Machine

machine [31]). We took an off-the-shelf tea machine that is internally driven by an Atmel AT89C2051 microcontroller with 2KBytes flash and built a mobile controller for it using R-OSGi. Since the microcontroller is not powerful enough to run Java and R-OSGi, we added a serial port to the board and implemented a plain RS232 protocol to give out status messages and to control the brewing. The tea machine is connected to and controlled by an external LinkSys NSLU2 (*Slug*) embedded linux device.

The tea machine can be spontaneously remote-controlled by PDAs. When a user's PDA comes within range of the machine, it can download the corresponding R-OSGi presentation (Figure 6(c)) and then control the machine (Figure 6(b)). The controlling is done by method invocations on the service that is running on the Slug. This service interacts with the machine over the RS232 link. The status information received from the tea machine is transmitted to the presentation on the PDA as remote events. The current application allows notifications about the status of the tea machine to be sent per e-mail and also to a desktop machine by using R-OSGi over HTTP.

This use case demonstrates the potential of the concept of presentations. The current limitation of the approach is that the problem of different ratios and resolutions of the displays of mobile devices cannot be solved by using predefined AWT panels as R-OSGi does now. We will address this in future work and plan to extend R-OSGi's presentations to support a more declarative way of defining user interfaces that support adaptation to the end device.

8 Conclusions

R-OSGi allows distributed applications to be built using the same modularity features of OSGi, and allows existing OSGi applications to be transparently

distributed along module boundaries. R-OSGi maps partial failures of the distributed application as a whole onto local module unload events, and represents traditional distributed systems support functions like service location as existing OSGi registration services. Experience has shown this novel approach avoids the problems typically encountered by transparent distribution systems, and we argue that R-OSGi does not present failure patterns to applications that could not occur in the centralized case.

While R-OSGi exploits OSGi's bundle concept to achieve this goal, it addresses significant further challenges. R-OSGi ensures consistency among shared classes by the use of type injection, and uses dynamic client proxy generation to allow even resource-constrained devices to provide services. R-OSGi is portable to all J2ME CDC profiles, is code compatible with Java back to Java 1.2, and runs entirely over a standard OSGi implementation. Despite these advantages, it is remarkably lightweight: R-OSGi has a file footprint of just 120 kBytes, slightly outperforms RMI in network tests, and is an order of magnitude faster than UPnP. Consequently, we argue that R-OSGi is an attractive approach to efficiently handle the complex structure of pervasive environments.

Acknowledgements

The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

References

1. OSGi Alliance: OSGi Service Platform - Release 4 (2005)
2. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Labs (1994)
3. Birrell, A., Nelson, B.J.: Implementing remote procedure calls. ACM Transactions on Computer Systems 2(1), 39–59 (1984)
4. Object Management Group: The Common Object Request Broker: Architecture and Specification. 2nd edn. (1995)
5. Brown, N., Kindel, C.: Distributed Component Object Model Protocol – DCOM/1.0 (Expired Internet Draft). IETF (1998)
6. Mullender, S.J., van Rossum, G., Tanenbaum, A.S., van Renesse, R., van Staveren, H.: Amoeba: A Distributed Operating System for the 1990s. IEEE Computer 23(5), 44–53 (1990)
7. Shapiro, M., Gourbant, Y., Habert, S., Mosseri, L., Ruffin, M., Valot, C.: SOS: An Object-Oriented Operating System - Assessment and Perspectives. Computing Systems 2(4), 287–337 (1989)
8. Hunt, G.C., Scott, M.L.: The Coign Automatic Distributed Partitioning System. In: OSDI 1999. Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (1999)

9. Tilevich, E., Smaragdakis, Y.: J-Orchestra: Automatic Java Application Partitioning. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 178–204. Springer, Heidelberg (2002)
10. Corwin, J., Bacon, D.F., Grove, D., Murthy, C.: MJ: A Rational Module System for Java and its Applications. In: OOPSLA 2003. Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 241–254. ACM Press, New York (2003)
11. Waldo, J.: The Jini architecture for network-centric computing. Communications of the ACM 42(7), 76–82 (1999)
12. UPnP Forum: Universal Plug and Play Device Architecture (2000)
13. Paremus: The Newton Project (2006), http://newton.codecauldron.org
14. The Eclipse Foundation: Eclipse Project (2001), http://www.eclipse.org
15. Apache Foundation: Apache Felix (2005), http://incubator.apache.org/felix
16. Gatespace Telematics SA: Knopflerfish OSGi (2003), www.knopflerfish.org
17. Rellermeyer, J.S., Alonso, G.: Concierge: A Service Platform for Resource-Constrained Devices. In: Proceedings of the EuroSys 2007 Conference (2007)
18. Apache Foundation: Apache Tomcat (2006), http://tomcat.apache.org
19. Howes, T.: RFC 1960: A String Representation of LDAP Search Filters. IETF (1996)
20. Kriens, P., Hargrave, B.: Listeners considered harmful: The "whiteboard" pattern. Technical report, OSGi Alliance (2004)
21. Guttman, E.: Service Location Protocol: Automatic Discovery of IP Network Services. IEEE Internet Computing 3(4), 71–80 (1999)
22. Veizades, J., Guttman, E., Perkins, C.: RFC 2165: Service Location Protocol. IETF (1997)
23. Guttman, E., Perkins, C., Veizades, J.: RFC 2608: Service Location Protocol v2. IETF (1999)
24. Rellermeyer, J.S.: jSLP project, Java Service Location Protocol (2005), http://jslp.sourceforge.net
25. Goland, Y.Y., Cai, T., Leach, P., Gu, Y., Albright, S.: Simple Service Discovery Protocol (Expired Internet Draft). IETF (1999)
26. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A Code Manipulation Tool to Implement Adaptable Systems. Technical report, France Telecom R&D (2002)
27. Haumacher, B., Moschny, T., Philippsen, M.: The Javaparty Project (2005), http://www.ipd.uka.de/JavaParty
28. Demuru, M., Furfari, F., Lenzi, S.: The Domoware UPnP service for OSGi (2005), http://domoware.isti.cnr.it
29. Sun Microsystems: Web Service Preformance. Comparing Java 2 Enterprise Edition (J2EE platform) and.NET Framework (2004)
30. Sun Microsystems: J2me Personal Profile for Zaurus 2002, http://java.sun.com/developer/earlyAccess/pp4zaurus
31. The University of Cambridge Computer Laboratory: The Trojan Room Coffee Machine (1991), http://www.cl.cam.ac.uk/coffee/coffee.html