Using Non-Java OSGi Services for Mobile Applications *

Jan S. Rellermeyer Michael Duller Gustavo Alonso
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland
{rellermeyer, dullerm, alonso}@inf.ethz.ch

ABSTRACT

OSGi is a popular software engineering approach for building complex software systems out of reusable Java modules. The OSGi framework describes an infrastructure for managing the modules at runtime and composing loosely-couples applications through services. With R-OSGi, we have extended the scope of OSGi from single applications to networks of applications consisting of distributed services. However, the OSGi model is entirely Java-centric and not trivially transferable to other languages, without losing generality and interoperability. In this demonstration, we show an application which uses OSGi services written in languages like TinyOS.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, Packages*; H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Design, Languages

Keywords

OSGi, R-OSGi, Middleware, Sensor Networks

1. INTRODUCTION

Modular software systems have gained large momentum in the past. One reason is the growing complexity of modern software systems. Systems built out of modular components are easier to control and to maintain. Another reason is the increased availability of reusable open-source implementations for common problems. A modular approach greatly simplifies the integration of different out-of-the-box components. Although the modular design has its origin in software architecture, it has been shown, for instance, in [5], to be a feasible starting point for building distributed systems by distributing a system along the boundaries of modules.

1.1 OSGi

OSGi [8] is an open standard that describes how to develop software modules in Java. Additionally, it specifies a runtime infrastructure, the so-called FRAMEWORK, to load new modules and manage the lifecycle of running modules at runtime. Due to the lifecycle management, OSGi was initially popular with embedded and other long-running systems. Recently, OSGi has obtained more attentiveness in the area of large, server-side applications. By making the dependencies between modules explicit, OSGi can dynamically compose the modules at runtime without losing the type consistency within the VM. This, however, typically leads to tight coupling between dependent modules. To overcome this limitation and to allow for more dynamic compositions, OSGi facilitates the loose coupling of modules through services. Every plain Java object can be a servive when it is published under a set of service interfaces. Clients can retrieve services from a central service registry by the name of the interface and, optionally, filtered by predicates over service attributes.

1.2 R-OSGi

R-OSGi [9] is the conceptual extension of the OSGi model to distributed systems. Through a transparent middleware, OSGi services from remote peers can be accessed as if they were local services. R-OSGi is capable of building dynamic proxies for remote services which show the same behavior and are synchronized with the state of the original service on the other machine. Therefore, it is possible to rapidly prototype systems in a local setting and without regarding distribution. The resulting modules can subsequently be deployed to different machines and connected through R-OSGi remote service invocations. Since OSGi does not pose any restriction on services and in fact accepts potentially every Java object to become a service, R-OSGi has to deal with the same miscellaneousness that is expressible in Java. It hence uses the Java serialization protocol in certain stages of the protocol, for instance, to ship the arguments for a remote service call.

*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.

MiNEMA'08, March 31-April 1, 2008, Glasgow, Scotland

1.3 Language independent OSGi

In order to use the principles of OSGi for new kinds of applications outside the Java microcosms, it deems eligible to have OSGi-like interactions with modules authored in languages others than Java. Preferably, these non-Java modules should be able to seamlessly communicate and interoperate with traditional OSGi frameworks. Since non-Java modules can typically be expected to run at least in other address spaces than an instance of a Java VM, the use of the R-OSGi protocol can be an option even when all modules reside on the same machine. A particular challenge of language-independent OSGi is indeed to integrate different languages without resorting to the least common denominator of all languages. This has traditionally been pursued with language-independent middleware like CORBA [7], which is—without further effort of custom type mapping—restricted to a defined set of supported types. More recently, web service [1] approaches like, for instance, SOAP [11], have reimplemented this idea by using the XML type system. For a sophisticated model like OSGi, however, these approaches oppose to the idea of perpetuating the full expressiveness that OSGi facilitates within the Java language.

2. SERVICES IN OTHER LANGUAGES

In traditional systems, the main load caused by an interaction between a service and a client rests with the service provider side. It is thus typically assumed that the service provider is a more powerful machine than the client, as in traditional client/server settings. R-OSGi, in contrast, has the property that the requirements for offering a service are smaller than for consuming a service, which might appear counterintuitive at first. The reason is that R-OSGi generates the dynamic proxies on the client side. This requires the availability of certain language features, like, for instance, dynamic loading and reflection. Conversely, a service can be provided with a minimum of language support. For instance, in [10], we have implemented OSGi services in languages like C, or even on sensor nodes running TinyOS [4]. As mentioned before, R-OSGi requires the serialization of, for instance, arguments of service method calls and this has traditionally been a reason to assume that such protocols can only be used among Java peers.

The reason why R-OSGi services are possible without a Java VM involved is that a service provider has to understand only as much of the Java serialization protocol as it actually needs. This involves, for instance, arguments of service methods and types used in service properties. However, these types are known at compile time. Hence, the serialization support for these types can be generated in any language if the corresponding Java types are known. The resulting custom serialization is bundled together with an R-OSGi protocol handler, a Java interface of the service as a *blob*, and the service implementation. Through the abstraction of interfaces in OSGi, the actual implementation of a service does not make a difference to clients of the service. With the same consequence, a service in R-OSGi can be written in any language as long as it behaves from outside like a Java service implementing the corresponding service interface.

In the subsequent sections, we briefly present implementations for two non-standard Java platforms, CLDC and TinyOS.

2.1 CLDC

The connected limited device configuration is the smallest configuration of Java platforms and runs on the KVM [12]. Typically, additional packages are bundled together with CLDC base packages into *profiles* like the MIDP that is available on most recent mobile phones nowadays and runs application bundles referred to as *MIDlets*. Due to limitations of the KVM, it is not possible to run OSGi on top of CLDC. In order to make services provided by CLDC devices available to other devices running OSGi and R-OSGi, we overcome the missing custom classloading and serialization support by automatically generating specific MIDlets. These MIDlets contain a stripped-down implementation of the R-OSGi protocol, custom serialization and deserialization code, and the interfaces and implementations of the services. Figure 1 illustrates how these resources are assembled into a MIDlet.

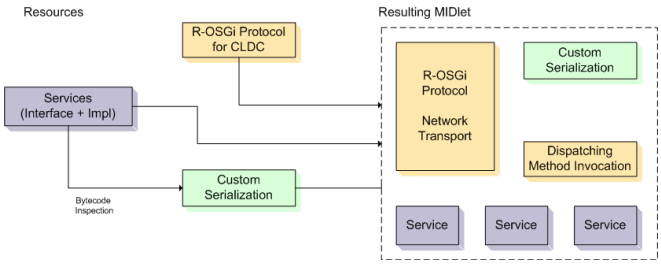

Figure 1: Generation of MIDlet.

2.2 TinyOS

Similar to the CLDC approach, we enable access to services provided by TinyOS based sensor nodes. A stripped-down version of the R-OSGi protocol implemented in NesC is assembled with custom serialization and deserialization code and the actual service implementations into a TinyOS application.

NesC itself also employs the concept of interfaces as well as it supports parametrized calls which can be leveraged for assembling multiple services into a TinyOS application. By adhering to a common interface we defined, the services can be accessed uniformly from the stripped-down R-OSGi protocol and dispatching of calls to a particular service happens directly through parametrized calls.

3. DEVICE-SPECIFIC TRANSPORTS

In the original implementation, instances of R-OSGi relied on TCP/IP transport for communication. Small and embedded devices, however, often do not provide TCP/IP transport since it significantly increases memory and processing footprint. In addition, the flexibility and powerful features that TCP/IP provides on top of the network interface are often not required when interacting with small devices. Hence we enhance R-OSGi to be able to use alternative transport protocol, that are more suitable for small devices. Below, we introduce two alternative transport protocols for R-OSGi.

3.1 Bluetooth

We implement native Bluetooth [2] transport for R-OSGi on top of the L2CAP transport layer, which provides reliable

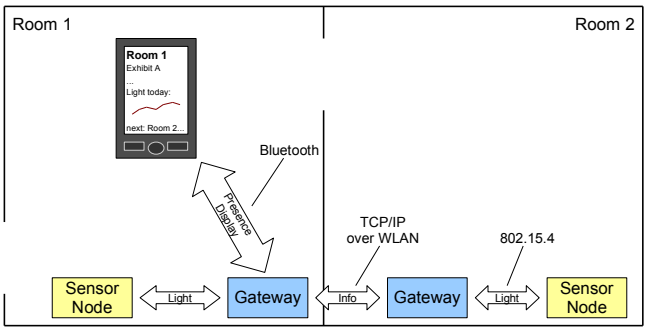

Figure 2: Demonstration Setup.

end-to-end communication. Besides the actual transport, we also facilitate service discovery using Bluetooth's native service discovery mechanism SDP [3]. For this purpose, we map OSGi service interface names and their properties to SDP entries in a way that directly allows to search over SDP UUIDs when searching for a particular service. The details of this mapping are presented in [10].

3.2 802.15.4 Transport

Many sensor node platforms, including the Tmote Sky [6], feature a 802.15.4 compliant radio. We implement a simple transport protocol on top of TinyOS' 802.15.4 radio stack that uses acknowledgements, timeouts, and retransmissions to provide reliable end-to-end communication. Furthermore, we implement the same transport protocol on TinyOS' Java SDK libraries and enable it as transport for R-OSGi. Thereby we can use it for communication with services implemented on TinyOS as well as use it as transport protocol between two instances of R-OSGi.

4. DEMONSTRATION

We will use the technologies introduced in the previous sections in the demonstration. The scenario is set in a building where visitors walk through the rooms and carry their mobile phones with them. For the demonstration, which is schematically depicted in Figure 2, we use two rooms, without limiting generality.

The mobile phones run a MIDlet on top of their CLDC Java environment that provides two services that can be discovered and accessed through the mobile phones' Bluetooth radio. One service is a presence service which simply identifies the user of the mobile phone and the second service is a display service which provides access to the phone's display, i.e., it allows to display content on the mobile phone.

In each room, a sensor node is deployed which runs the TinyOS implementation of a light service as defined in Listing 1. The method `getLight` accepts an output stream and an interval. When the method is called on the Tmote, it will start sampling its light sensor every *interval* seconds and write the sampled value to the output stream *os*. The service can be accessed using the 802.15.4 transport.

The third entity involved in the scenario is a room gateway, of which we deploy one in each room. The gateways run a full Java VM hosting an OSGi framework with R-OSGi and a module that consumes and orchestrates the services provided by phones and sensor nodes. Furthermore, it interacts with the other gateway.

Listing 1: Light service interface.

```
public interface LightService {
    public void getLight(OutputStream os,
        int interval);
}
```

When a visitor wanders through the rooms and thus comes within the range of the Bluetooth radio of a room gateway, the gateway discovers the presence service of the mobile phone and can identify the user. If this user subscribed to receive information on its mobile phone, the gateway will access the display service provided by the user's phone and display static information about the room (e.g., exhibits) as well as dynamic information gathered from the sensor nodes like, e.g., the light sampled in this room during the day. Additionally, when the gateway can interact with gateways of neighboring rooms, it will also display abstract information about the neighboring rooms.

5. REFERENCES

[1] G. Alonso, F. Casati, H. A. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, 2004.

[2] Bluetooth Special Interest Group. Bluetooth. `http://www.bluetooth.com`, 1998.

[3] E. Gryazin. Service discovery in bluetooth.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, 2000.

[5] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI 1999)*, 1999.

[6] Moteiv. Tmote sky. `http://www.moteiv.com/`, 2005.

[7] Object Management Group, Inc. Common Object Request Broker Architecture: Core Specification, 2004.

[8] Open Service Gateway Initiative. `http://www.osgi.org`.

[9] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed Applications Through Software Modularization. In *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference*, 2007.

[10] J. S. Rellermeyer, M. Duller, K. Gilmer, D. Maragkos, D. Papageorgiou, and G. Alonso. The Software Fabric for the Internet of Things. In *Proceedings of the 1st International Conference on the Internet of Things*, 2008.

[11] Simple Object Access Protocol. `http://www.w3.org/TR/soap/`.

[12] Sun Microsystems, Inc. J2ME Building Blocks for Mobile Devices: White Paper on KVM and the Connected, Limited Device Configuration (CLDC), May 2000.