

# Services Everywhere: OSGi in Distributed Environments \*

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

## ABSTRACT

*Distribution is increasingly becoming an important issue in both enterprise applications and mobile computing. OSGi itself has only rudimentary support for distribution, in forms of interfaces for interaction with Jini (R3) or UPnP (R3 + R4) infrastructures. When it comes to interconnecting different OSGi frameworks, there are only few solutions so far. In this paper, we present these existing solutions and compare the different approaches with our own R-OSGi. The goal of our open source project is to provide a seamless and non-invasive middleware for accessing remote services in OSGi frameworks. We explain the basic design principles of R-OSGi, such as transparent service access and spontaneous interaction, and briefly mention the internal structure and techniques used in R-OSGi, such as service discovery and smart proxies.*

## 1. INTRODUCTION

The OSGi framework is nowadays a well-established approach for solving the problem of modularization of Java applications. Software can be modularized into independent bundles interacting through services. Services reduce the coupling of bundles by defining service interfaces. Whereas the interface of a service is the shared knowledge between interacting bundles, the concrete implementation of a service remains a black box. This not only reduces link-level dependencies to the types used in the interface but also allows to exchange the implementation of a service at runtime.

In the context of networked environments, the service-oriented design of OSGi is ideal for building distributed applications. The strict distinction between interface and implementation and the loose coupling between components make it possible to access services residing on other ma-

\*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.

Some rights reserved.

This publication is licensed under a Creative Commons Attribution-NonCommercial 2.5 License; see <http://creativecommons.org/licenses/by-nc/2.5/> for further details.

EclipseCon 2007 March 5–8, 2007, Santa Clara, CA.

chines if a feasible solution for the integration of remote services into the OSGi framework can be achieved.

## 2. EXISTING APPROACHES

Two ways of dealing with remote devices are already described in the OSGi specifications. The R3 specifications introduced services for interaction with UPnP and Jini infrastructures. Although the Jini service is no longer contained in the R4 specifications, it is discussed here for the sake of completeness. In the following section, we briefly characterize the two technologies and describe the implications in the context of OSGi services.

### 2.1 Jini

The basic idea behind Jini [13, 10] is to provide a runtime infrastructure for federating distributed services within a network. The central point in any Jini communication is the *Lookup Service* (LUS). In a first step (*discovery*), peers have to find a LUS. Either they have preconfigured knowledge about the LUS or they perform multicast discovery to find one within their subnet. The discovery returns a marshalled `net.jini.core.lookup.ServiceRegistrar` object which is a Java RMI proxy for the LUS. In the second step (*join*), a peer publishes all own services to the LUS. In the *lookup* step, peers can find services that match specific service interfaces, attributes, or that have a specific known service ID.

Although the Jini specifications do not impose a specific protocol for the communication with the services itself, the Sun implementation requires RMI for contacting the Lookup Server. In many existing deployments of Jini, people use RMI for all communication. This violates the generality of the OSGi Jini Service since RMI is not part of any standard execution environment. In practice, especially small mobile and embedded devices often don't have RMI.

The OSGi service described in the R3 specifications defined the export of OSGi services to Jini and the import of Jini services into OSGi. However, in practice, implementations of this service turned out to be fairly invasive. One reason is that the attributes in Jini are arrays of Objects implementing the *Entry* interface whereas in OSGi, attributes are modeled as key/value pairs of arbitrary types. There is no straightforward way of mapping OSGi properties to Entries and vice versa to preserve the sophisticated matching capabilities of OSGi. Additionally, the possibility to publish a service only under specific interfaces is not given in Jini. A service object is always accessible under all its interfaces, there is no way of selectivity. The requirement of a cen-

tral Lookup Server furthermore limits the usability of Jini in patterns of spontaneous interaction and imposes a significant amount of configuration and managed infrastructure.

## 2.2 UPnP

UPnP [12] is an industry standard for service-oriented interaction with consumer devices. It is frequently used in ubiquitous environments. UPnP is based on well-established standards like HTTP, XML, and SOAP. The UPnP Forum specifies *device profiles* for common applications. Each device can specify one or more services which consist of *actions* (equivalent to service methods in OSGi) and *events* for asynchronous notification of the clients. Furthermore, UPnP defines an addressing algorithm derived from Zeroconf [4] to allow spontaneous interaction in unmanaged network environments and a service discovery based on SSDP [2]. SSDP uses multicast HTTP messages for service advertisements and searching specific services in the network. Devices and services are described in description files which are XML-encoded and comply to an UPnP-specific schema. From these descriptions, clients can derive the information about the service interface and invoke actions. The invocation of actions is based on SOAP messages.

The OSGi specifications define a *UPnPDeviceService* to facilitate interaction with UPnP-enabled consumer devices. However, the services created by the UPnP base driver are highly UPnP-specific and thereby limited. First, UPnP services can only make use of a subset of the standard SOAP data types. It is furthermore not possible to pass complex Java objects as arguments for UPnP actions since extensive type mapping is not supported. Events can be nicely integrated into the OSGi *EventManager* service. However, UPnP does not allow to subscribe to specific events. Clients always have to subscribe to all events. Additionally, the high verbosity of the XML format and the SOAP envelopes lead to a significantly decreased performance compared to binary protocols like Java RMI.

In conclusion, OSGi is an ideal environment for building UPnP devices since the *UPnPDeviceDriver* is much more intuitive to use and well-integrated into Java than many other UPnP stacks. However, UPnP is not ideal for distributing OSGi applications since it severely limits the richness of OSGi services.

## 3. DISTRIBUTED OSGI

The experience with UPnP and Jini has shown that none of the two approaches is ideal when it comes to the federation of distributed OSGi frameworks. The challenge of bridging independent OSGi frameworks is to model the interaction between the services of the individual peers as well as taking care of module-layer dependencies among the providing bundles. An ideal approach would be to have a middleware layer that allows multiple peers with local OSGi frameworks to be federated in such a way that they behave like one large OSGi framework. For doing so, some key requirements can be stated:

- *Transparency*: For each local OSGi framework, the distributed character of the whole federation should be masked. Remote services should be accessible as if they were locally present in the framework.
- *Non-Invasiveness*: The middleware should not impose any restrictions on OSGi services. In particular, it

should be possible to use existing bundles and services in distributed setups without any need for modification.

- *Consistent Behavior*: The default behavior of OSGi services should not be influenced by the distribution. The perspective of a remote peer on a service has to be the same as that seen by an entity within the local framework.
- *Spontaneous Interaction*: It should be possible to federate OSGi-enabled peers in a spontaneous way, avoiding any preconfiguration or management.
- *Statement of Supply and Demand*: To allow for massively distributed setups, for instance, the Internet, it has to be ensured that a certain level of selectivity for the interaction is possible. It has to be avoided that a peer is overwhelmed by the number of available services and thereby rendered inoperable.
- *Generality*: The OSGi specifications cover a large range of computing devices. In particular, the lightweight design of the OSGi architecture allows frameworks to run on very resource-constrained devices. Therefore, a middleware for distribution should not limit the configurations where OSGi can be used.

## 4. R-OSGI

We have started to address these requirements in our ongoing research project *R-OSGi*<sup>1</sup>. The following section discusses the fundamental design principles of R-OSGi and how they are intended to cope with the challenges arising from distributed OSGi setups.

### 4.1 Service Discovery

As stated before, spontaneous interactions is a key requirement, especially when it comes to mobile devices interconnected by networks with little management, such as ad-hoc networks. In such an environment, no preconfigured knowledge about the location of particular services can be assumed. The same is true for consumer electronic networks, for instance, in smart home environments. These networks have to be extensible without requiring an unreasonable amount of technical education from the end-user. One possible way out is the use of service discovery protocols. This allows peers to explore their environment and find useful services to extend their own functionality.

In R-OSGi, we make use of the Service Location Protocol (SLP) described in RFC 2608 [5, 3]. This protocol is well-established in the area of system administration and has several advantages over comparable approaches. First, it is a binary protocol with a small message size, thereby making it suitable for setups with restricted network bandwidth and resource-constrained devices. Second, it can operate with a central service broker (DA, Directory Agent) and in peer-to-peer setups by using a multicast convergence strategy. Last and most important, the notion of services in SLP is very close to the one used in OSGi. SLP services are described by a *ServiceURL*, consisting of a service type and a URL address. Furthermore, SLP services can have key/value attributes and requests support LDAP-style filter predicates

<sup>1</sup><http://www.iks.inf.ethz.ch/projects/>

on these attributes. It is hence possible to create an unambiguous one-to-one mapping from OSGi services to SLP services.

R-OSGi uses such a mapping to announce OSGi services to surrounding peers. To avoid scalability and security issues, some entity within the framework has to explicitly mark a service to be released for remote access. This is done by setting a specific remote property in the service properties. To preserve a non-invasive character, it is not required that this is done by the service providing bundle itself. R-OSGi also facilitates adapter bundles to make surrogate remote registrations. The registration of services for remote access is referred to as *statement of supply*.

Bundles can register *Discovery Listeners* to express their demand for services implementing a certain interface and/or matching a specific filter predicate. This fully complies with the known patterns from OSGi frameworks. The registration of discovery listeners forms the *statement of demand*.

R-OSGi is in charge of exploring the environment by the use of the SLP protocol and to match supply and demand. If such a match is detected, the corresponding discovery listener that has stated the demand is called and the application can explicitly decide whether a connection to the supplying peer should be established.

## 4.2 Network Channels

Connections in R-OSGi are implemented by network channels. Each channel is a one-to-one connection between two peers, as shown in Figure 1. Upon establishment of a network channel, the two peers exchange symmetric leases that contain their statements of supply in terms of offered services and their demands in terms of event topics the peer is interested in. The details about remote events are discussed in section 4.6. Different to Jini, leases in R-OSGi have no built-in expiration. By design, they only have to be unilaterally renewed when either the supply or the demand has undergone a change. Otherwise, leases are valid unless the channel is closed, either intentionally or due to a permanent network failure. Besides service discovery, peers can also directly connect to known service providers. One possible application is wide-area distribution over the Internet where service discovery is not feasible.

By default, network channels are implemented as persistent TCP connections. This allows maximum performance for service invocation and event delivery since the overhead of the TCP handshake is only required once. The extensible design of R-OSGi, however, facilitates to plug in alternative protocols and network transport. One example for this is the implemented HTTP(S) network channel discussed in Section 4.8.

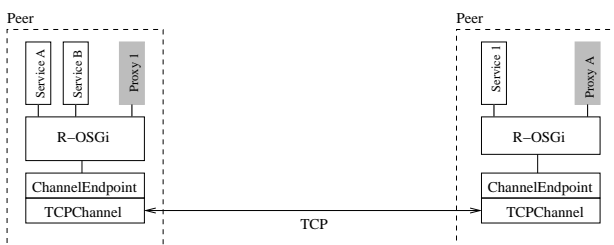


Figure 1: R-OSGi Network Channels

## 4.3 Proxy Generation

Once the application has established a network channel to a peer, it can decide to fetch the service matching its requirements. The fetching basically involves the transmission of the service interface and the service attributes. This is the minimum amount of information required to access a remote service by building a service proxy. Proxy generation is implemented using bytecode manipulation based on the lightweight ASM library [1]. The service interface is taken and a class is created on the fly which implements every method described in the interface as a remote method invocation. Additionally, the class implements *BundleActivator* to integrate into the OSGi lifecycle management. During registration of the remote service, the type injections have been determined. These are described by the minimal set of types that have to be injected in order to make a service proxy operable, providing that the proxy is allowed to import any package that is also imported by the original service. The injections are transmitted as part of the fetching and materialized together with the proxy class into a proxy bundle. This proxy bundle now provides the remote service to the framework and redirects every method call to the original service.

On a first view, it might appear eligible to overcome the limitation that under certain conditions, the proxy bundle has to import some packages to make the service interface resolvable. Instead, it would be possible to additionally inject all imported packages to make the proxy bundle self-contained. However, for real applications, this potentially involves a large number of classes and packages. Since too extensive injection can significantly increase both the network bandwidth consumption and the footprint of the proxy bundle, this strategy is not followed by R-OSGi.

## 4.4 Method Invocation

Method invocation in R-OSGi uses its own message-based protocol. The R-OSGi service features a generic method which is called by every proxy method in a reflective style. This successively leads to the transmission of a specific message that transfers the description of the called method (service URL and method signature) together with the arguments of the call to the remote peer. On the service provider side, the original service method is then called. This either succeeds and possibly results in a return value, or an exception might be thrown. In either case, a response message is generated and sent back to the calling peer. The proxy method now either returns the result or throws the deserialized exception object. This models the exact behavior that a local service method would have.

One obvious limitation is that the formal parameter of remote services have to be serializable. This is known from many other remote invocation mechanisms and can currently not be easily avoided in R-OSGi. Under some circumstances, it is, however, possible to add the serializability to existing classes by applying load-time AOP.

Since Java already contains RMI for remote invocation, it has to be justified not to make use of it in R-OSGi. Several reasons have led to this design decision. RMI is not available on every CDC platform. Even if it is available, the performance of these implementations is often unsatisfying. Different to ordinary remote objects, service objects in OSGi have a well-defined lifecycle, thereby not requiring the overhead of mechanisms like distributed garbage collection (DGC).

Furthermore, the well-defined lifecycle allows to keep the persistent connection open for a longer period, thereby reducing the overhead of additional TCP handshakes. In our preliminary experiments, the remote invocation of R-OSGi is in average faster and scales better than the highly optimized RMI implementation in Java 5.

## 4.5 Smart Proxies

By using service proxies, the major load of the service remains on the service providing peer. This is sometimes not the favored behavior, especially when the service provider is, for instance, a resource-constrained ubiquitous device offering services to a potentially large number of clients. R-OSGi allows to use smart proxies to overcome this and ship parts of the service code to the client. In the properties of the service registration used to publish the remote service, an additional abstract class implementing the service interface can be defined to be the smart proxy. This smart proxy class is transferred to the client. All implemented methods remain untouched, whereas the abstract methods are implemented as remote invocations in the usual manner. With an elaborated design, it is possible to solve complex problems where services perform parts of the tasks locally using the resources and configurations of the clients. A simple example for this pattern can be seen in Listing 1.

**Listing 1: Smart Proxy Example**  
**package** sample.service;

```
import sample.api.ServiceInterface;

public abstract class SmartService
    implements ServiceInterface {

    public void local(String s) {
        // do some local operations
        System.out.println("Local_operation");
        remote(s);
    }

    public abstract void remote(String s);
}
```

## 4.6 Remote Events

Events are a known technique from the UPnP world to allow for asynchronous notification. Since OSGi R4 features a standard way of handling events, R-OSGi has been integrated to make use of the *EventAdmin* service. On each peer, the topic space is determined from the intersection of all registered *EventHandlers*. As part of each channel endpoint, an additional event handler is registered which forwards all topics matching the topic interest of the other side of the channel. Thus, bundles making use of the event handling infrastructure of OSGi can be transparently distributed. The registration of a new event handler on an R-OSGi enabled peer that extends the topic space leads to an update of the leases and to a modification of the event handler registered by the channel endpoint.

## 4.7 Presentations

An additional feature of R-OSGi is the transmission of

presentations. The idea of a presentation is to ship a user interface together with the service. This user interface can be displayed using the R-OSGi *ServiceUI* bundle. The use case for presentations is the controlling of smart devices. Instead of requiring the user to install a device driver on each controller device, R-OSGi and presentations can be used to let the device itself provide a user interface for interaction. In spontaneous setups, e.g., a PDA can be used to control an arbitrary number of previously unknown devices that are dynamically discovered. The *IKS Lego Mindstorms Robots* described in section 5.1 are an example for this idea.

Currently, presentations are transmitted as prefabricated AWT Panels. However, the diversity of end devices and the different capabilities, for instance, in terms of display sizes and resolutions let this approach appear not really feasible for the general purpose. As part of future work, we intend to address this issue and come up with a more declarative way of describing user interfaces that can be rendered on the client device with respect to the particular capabilities of the hardware.



Figure 2: R-OSGi Presentation of a Smart Device

## 4.8 Alternative network channels

In massively distributed setups and in wide area networks, it is sometimes a requirement that the network protocol is able to pass firewalls. R-OSGi therefore allows to plugin alternative channel implementations which piggyback the native R-OSGi messages. As part of our research project, we have implemented such a channel for HTTP(S). The idea is to provide *NetworkChannelFactories* that can be registered in a whiteboard manner to be responsible for a specific set of protocols. Whenever a connection using a non-standard protocol is requested, the corresponding external factory is called to return a custom *NetworkChannel* object. It is even possible to transparently wrap completely different (e.g. non-IP-based) ways of network transport like, for instance, Bluetooth. Service providers have to additionally provide a bridge which accepts incoming connections of the custom protocol or transport type and forwards the piggybacked R-OSGi message to the R-OSGi service. An example

setup is given in Figure 3. A simple API allows to rewrite all ServiceURLs appearing in R-OSGi messages to restamp them to the address and port of the bridge.

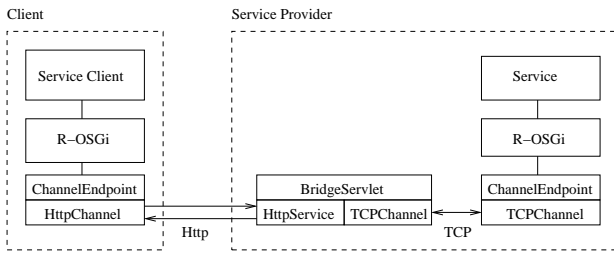


Figure 3: R-OSGi over HTTP

The interesting property of the HTTP(S) integration is the constraint that the protocol is entirely based on a request/response pattern whereas R-OSGi allows asynchronous callbacks by the server-side-equivalent through events. This problem is known from web services and typically solved by requiring the client-side-equivalent to also provide a server part that handles the callback. In R-OSGi, we have solved the problem differently. The bridge servlet on the service provider side catches the lease message and replies with a multipart-encoded response that is not closed unless the lease expires. This delayed response allows to send following events as new chunks of the response. This solution is compliant with the standard and works with both firewalls and client-side NAT.

#### 4.9 Alternative policies

Although the use of proxy bundles in some flavor is the default behavior of the system, it also supports alternative ways of distributing services. Currently, also whole bundles can be migrated to remote peers. For the moment, this only involves stateless migration of bundles but it is part of future work to integrate this policy with the *ConfigurationAdmin* to allow for stateful migration of OSGi bundles. One possible use case for this is adaptive load balancing for applications of server-side OSGi.

### 5. EXAMPLES

To further explain the powerful features of R-OSGi, the following section discusses some example applications that have been implemented using distributed OSGi.

#### 5.1 IKS Lego Mindstorms Robots

The Lego Mindstorms Robots (Figure 4) are an ongoing student lab project which is running for several years now. The robots are controlled by either an iPAQ, or the most recent version by a Linksys NSLU2 (Slug). The controller device communicates with the Lego RCX and transmits macros which are then executed. The robots can follow the lines painted on the floor and execute simple tasks like moving to the next junction, turning, etc. In our recent setup, the Slug runs Java, the Concierge [9] OSGi framework and the R-OSGi service. All controlling is performed by a *RobotDevice* service (Listing 2) which can be transparently accessed by end devices connected through 802.11 WLAN. The service has an attached presentation which can be displayed by the *ServiceUI*. We use PDA devices to spontaneously discover robots, connect to the robots, and remote

control them. Each PDA can control multiple robots and enqueue tasks that they perform in sequential order.

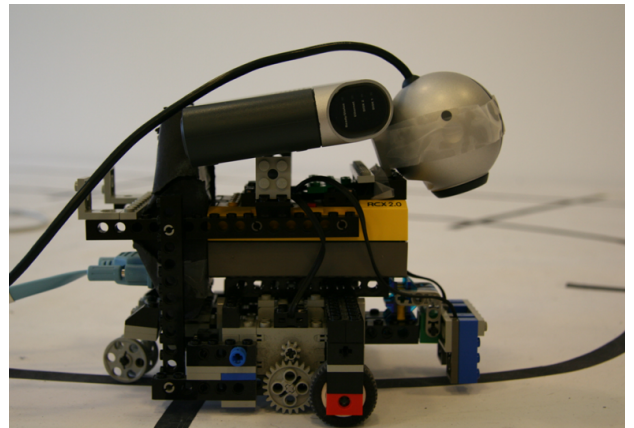


Figure 4: IKS Lego Mindstorms Robot

Listing 2: Robot Service Interface  
package ch.ethz.iks.robot.api;

```
public interface RobotDevice {

    public static final String ROBOT.TOPIC =
        "ch/ethz/iks/robot/";

    public void turnRight();

    public void turnLeft();

    public void goForward(int units);

    public String getQueue();
}
```

#### 5.2 Sensor Networks

The SwissQM project [6, 7] is a novel approach for wireless sensor networks based on the idea of having a virtual machine on each node. A gateway device connected to the sensor network generates bytecode programs from declarative queries on a gateway to allow users to gather data in a more intuitive and flexible way. The programs are disseminated in the network and return the data required to answer the queries. By having bytecode programs, processing can be pushed down into the network, thereby reducing the message complexity. One of the unique features of SwissQM is the possibility to include user defined functions (UDFs) in queries, similar to relational database systems.

In a development branch of SwissQM, we have used R-OSGi to integrate the gateway functionality into Eclipse. The high flexibility of the system required development support by a commonly used and intuitive IDE. An Eclipse plugin allows to write user defined functions and store queries on an arbitrary machine and interact with a remote SwissQM gateway. Furthermore, the gateway returns the result tuples to the development machine and the Eclipse plugin is able

to visualize the data (Figure 5). This facilitates rapid prototyping and debugging of queries and functions. R-OSGi is in charge of handling the communication between the Eclipse environment and the gateway machine, as shown in Figure 6. For the plugin, the gateway appears as an ordinary OSGi service. In this branch, SwissQM is able to be extended by support for parsing arbitrary high-level languages and compiles OSGi bundles from the queries and function code containing language-independent intermediate representations. This additionally allows to explicitly influence the lifecycle of individual queries and functions remotely through the Eclipse Plugin. To offer access from firewalled networks, the SwissQM Plugin uses R-OSGi over HTTP.

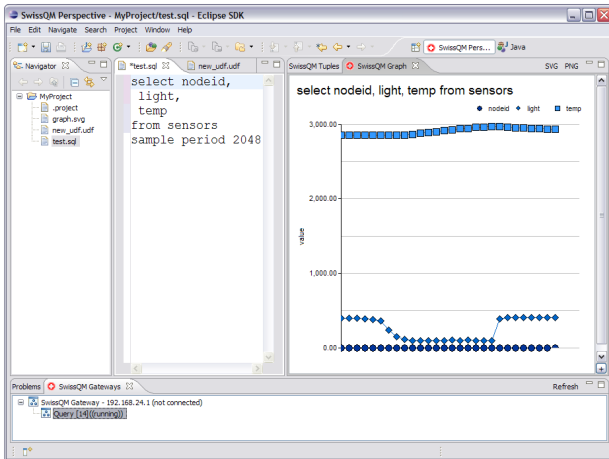


Figure 5: SwissQM Eclipse Plugin

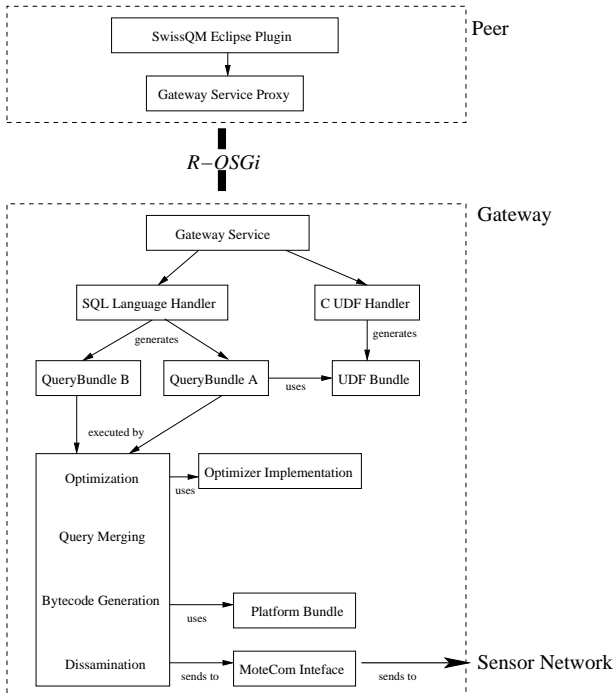


Figure 6: Architectural Overview

### 5.3 Fluid Computing

Traditional systems have a strong locality of data. Distributed applications typically store the shared data at well-defined locations, such as an application server or a database. The idea of fluid computing is to allow data to *flow* through the network. If a peer is disconnected, offline operations are recorded and reconciled with the global state if the peer returns to the network. In the *flowSGi* project [8, 11], we explore the possibilities to extend this kind of flexibility to whole applications. What goes far beyond previous projects in this area is the idea that also the application should be fluid, thereby allowing to start the work on one device and then seamlessly migrate the whole application together with its state and data to a different peer. Furthermore, it is possible to work concurrently from many peers on the same data with federated devices. In such a setup, each device provides its unique capabilities to perform a specific part of the application. R-OSGi is one of the key technologies required to achieve this. It is used to handle the communication on the middleware layer and to make the different peers interact and behave like one single entity.

### 6. REFERENCES

- [1] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A Code Manipulation Tool to Implement Adaptable Systems. Technical report, France Telecom R&D, November 2002.
- [2] Y. Y. Golland, T. Cai, P. Leach, Y. Gu, and S. Albright. *Simple Service Discovery Protocol (Expired Internet Draft)*. IETF, October 1999.
- [3] E. Guttman. Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing*, 3(4):71–80, 1999.
- [4] E. Guttman. Autoconfiguration for IP networking: enabling local communication. *Internet Computing, IEEE*, 5(3):81–86, 2001.
- [5] E. Guttman, C. Perkins, and J. Veizades. *RFC 2608: Service Location Protocol v2*. IETF, June 1999.
- [6] R. Müller and G. Alonso. Efficient Sharing of Sensor Networks. In *MASS*, 2006.
- [7] R. Müller, G. Alonso, and D. Kossmann. SwissQM: Next generation data processing in sensor networks. In *CIDR*, 2007.
- [8] J. S. Rellermeier. *flowSGi - a Framework for Dynamic Fluid Applications*. Master’s thesis, ETH Zurich, 2006.
- [9] J. S. Rellermeier and G. Alonso. Concierge: A Service Platform for Resource-Constrained Devices. In *Proceedings of the EuroSys 2007 Conference*, 2007.
- [10] Sun Microsystems. *Jini Specifications v2.1*, October 2005.
- [11] The flowSGi Project. <http://www.flowsgi.inf.ethz.ch>.
- [12] UPnP Forum. *Universal Plug and Play Device Architecture*, June 2000.
- [13] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.