Virtualization of Service Gateways in Multi-provider Environments

Yvan Royon, Stéphane Frénot, and Frédéric Le Mouel

INRIA Ares - CITI Lab - INSA Lyon
Bat. Leonard de Vinci, 69621 Villeurbanne cedex, France
{yvan.royon, stephane.frenot, frederic.le-mouel}@insa-lyon.fr

Abstract. Today we see more and more services being brought to connected homes, such as entertainment or home automation. These services are published and operated by a variety of service providers. Currently, each provider sells his own box, providing both connectivity and a closed service environment. The open service paradigm aims at mixing all services within the same box, thus opening the service delivery chain for home users.
However, open service gateways still lack important mechanisms. Multiple service providers can access and use the same gateway concurrently. We must define what this use is, *i.e.* we must define a notion of *user*. Also, service providers should not interfere with each other on the gateway, except if explicitly required. In other words, we must isolate services from different providers, while still permitting on-demand collaboration. By combining all these mechanisms, we are defining a multi-user, multi-service execution environment, which we call a virtualized service gateway. We implement part of these features using OSGi technology.[1]

Keywords: Virtual gateway, multi-user, service-oriented programming.

1 Introduction

During the last years, high speed connectivity to the home has evolved at a very fast pace. Yesterday, home network access consisted in bringing IP connectivity to the home. The services made available were common application-level programs, such as web or e-mail clients. Today, the operators are moving to integrating value-added services in their offer, mainly multicast TV and Voice over IP. These network-enabled services are provided by the same connectivity box, or by a dedicated set-top box. It is foreseen that in the next few years, both the number and the quality of available services will increase drastically: home appliances, entertainment, health care... However, these services would be developed, maintained and supervised by other parties, for instance respectively whitegoods manufacturers, the gaming industry, and hospitals. Until today, the entire service chain and the delivery infrastructure, including the home gateway,

[1] This work was partially supported by the IST-6thFP-507295 MUSE Integrated Project

are under the control of a single operator. Emerging standards push towards open solutions that enable both integration and segmentation of various markets such as connectivity, entertainment, security, or home automation. This approach implies that, on the single access point that connects the home network to the internet (*i.e.* the home gateway), many service vendors are each able to deploy and manage several services. Figure 1 shows the different parties involved in the network.

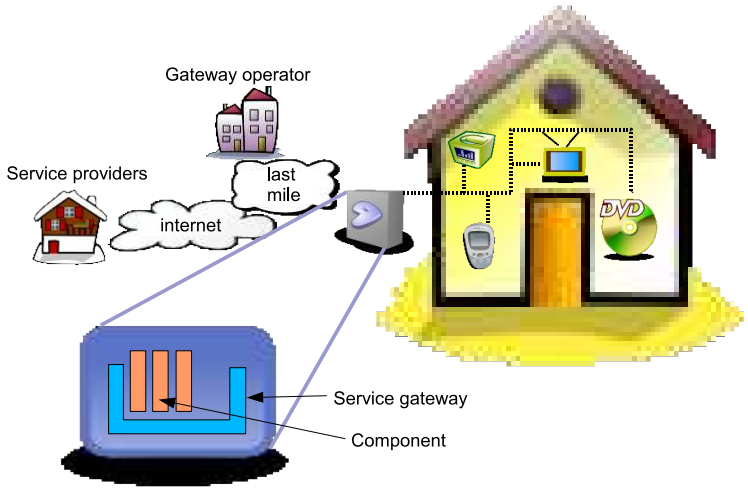

Fig. 1. Open service gateway

Current and ongoing service platform efforts enable multi-party service provisioning, but they still lack strong concepts and mechanisms to completely support it. For instance, no isolation of services from different parties has been defined. Also, no party-oriented management has been described.

We propose an isolation of services, depending upon which party, or *user*, deploys, manages and owns them. Consequently, we also define the notion of user in this context. By isolation, we mean that a service provider should only be able to "see" her own services on the common service platform. We represent this as a *virtual service gateway*. Each service provider owns and manages her own virtual gateway and the services it runs. All virtual gateways are run and managed by a unique *core service gateway*, typically hosted inside the home gateway (physical box), and operated by the home gateway provider.

By default, services within the same gateway are able to interact, while services in different gateways are not. However, our model adds the possibility to gain explicit, on-demand service cooperation between separate gateways.

The remainder of this paper is organized as follows. Section 2 describes ongoing works on multi-application Java environments. Section 3 defines the notion of user in this context and details the concepts of the virtual service gateway, *i.e.* a multi-service, multi-user, service-oriented environment. We also see how this technology model integrates with the business model descibed above. In section 4 we explain how we implement these concepts on top of an OSGi service plarform. Section 5 proposes perspectives for future works, and section 6 discusses and concludes this article.

2 Multi-application Java Environments

This paper focuses on Java-based environments. This choice is based on three facts. First, Java-based applications are usually architecture- and OS-agnostic, which is an interesting feature when using potentially different platforms such as home gateways. Second, Java's popularity keeps increasing, for server applications as well as handheld devices. Third, most service-oriented frameworks today are based on Java technology. In this section, we examine solutions to make Java a multi-application environment, essential for an open service gateway.

2.1 Current Java Environments

A standard Java Virtual Machine is a multi-thread-enabled but mono-application environment (figure 2). In order to run two applications, two JVMs are launched. In this case, the applications run independently, *i.e.* if they need to collaborate, they must access the operating system's communication facilities (*e.g.* TCP/IP network stack, file system...). We can see that the problems with this solution are both the overhead from running two JVMs, and the inefficiency of communications, even though there are proposals to limit these [1].

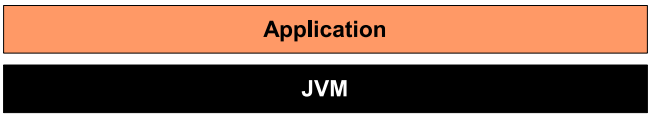

Fig. 2. Mono-application JVM

There are two kinds of responses to these insufficiencies: bringing multi-application capabilities to the JVM, or using an overlay on top of the JVM, *e.g.* a J2EE or similar application server.

2.2 Multi-application Java Environments

- Sun's Multi-tasking Virtual Machine [2] (figure 3), for instance, runs several Java applications, called *isolates* [3], in the same Java environment. Isolates

share class representation, so that only static fields are duplicated. Applications are instrumented using a resource management interface [4], for heap memory management in particular.

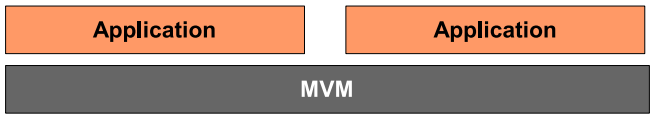

Fig. 3. Multi-task JVM

Rival proposals are Java operating systems [5] [6]. These mix the JVM with the operating system layer, and often come with multi-process capabilities.

- A last option is to add a multi-application-like functionalities using an overlay on top of the JVM (figure 4). The overlay is the single application that runs in the JVM, but it allows several pseudo-applications to run concurrently on top of it.

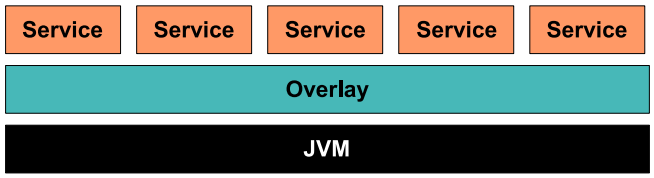

Fig. 4. Multi-service environment

2.3 Isolation Terminology

The term "isolation" may imply several kinds of mechanisms. We attempt here to basically classify them.

• The first family of mechanisms is namespace isolation. A namespace is a context for identifiers, and, in our case, for applications or services. Applications in different namespaces cannot "see" each other: this is an access right enforcement.
With Java technology, this namespace isolation may be achieved through the use of classloaders, or more advanced loading facilities such as the Module Loader [7] or MJ [8].

• The second family of isolation mechanisms concerns low-level resources. In a resource-isolated environment, applications are supposed to be protected from

one another. For instance, schedulers provided by operating systems allocate CPU slots for applications according to their priority. Recent Linux kernels also endorse out-of-memory kills, *i.e.* if a process endangers the whole system using too much memory, it gets killed. Such memory protection can be qualified as a reactive mechanism, versus proactive mechanisms. An example of proactive mechanism would be Xen's hypervisor [9].

There are two ways to combine namespace isolation and resource isolation. The first one is to complete namespace isolation with a reactive resource isolation, for instance by using a monitoring architecture that checks specific constraints such as CPU usage [10].
The second one is to build a combination of proactive, strong resource isolation and namespace isolation through the use of different virtualization techniques. Proposals such as Xen [9] or Denali [11] run multiple lightweight or full-featured operating systems on the same machine. Other attempts, such as Java *isolates*, provide an isolation API for Java applications.

2.4 Our Goals

The advantages of a modified runtime are performance (communications, memory sharing) and resource isolation (see below). Inversely, the advantages of overlays are their usability on any standard JVM, and their ease of development through their level of abstraction. Table 1 summarizes this comparison.

	Namespace isolation	Resource isolation	Perfomance optimizations	Uses a standard JVM	Easiness of integration[2]
Modified JRE	yes	yes	yes	no	intrusive
Overlay	no	no	no	yes	effortless

Table 1. Summary of Multi-application Java Environments

Our goal in this paper is to define a multi-user, service-oriented Java environment, without modifying the standard Java Runtime Environment. This implies that we use an overlay. Our first step is to add namespace isolation to the overlay solution. Then, we add a definition of users.

3 Towards Multi-user, Service-oriented Java Environments

In this section, we evoke the notion of service-orientation an dits benefits. We then detail the terms multi-user, through the definition of *core* and *virtual* service gateways. We explain on one hand how they should be isolated, and on the other

[2] Easiness of integration means the easiness of developing the environment itself, using it, and developing applications for it.

hand when and how they should be able to cooperate. Also, as seen before, services inside a service gateway are run on behalf of a remote service provider. This provider may need to manage its services; also we propose simple remote management interfaces for the different actors involved.

3.1 Service-oriented Programming

Service-Oriented Programming (SOP) is a paradigm that focuses on what a piece of code does. It is based on Object-Oriented Programming (OOP), which, by comparison, focuses on how data and processing are coded [12]. While OOP relies on ideas such as encapsulation, inheritance and polymorphism, SOP states that elements (*e.g.* components) collaborate through behaviors. These behaviors, also called services or interfaces, allow to separate what must be done (the contract) from how it is done (its implementation).

Some SOP solutions, such as Web Services, deal with the interoperability in communications. Others, such as the OSGi Service Platform [13] and Openwings [14], are centered on the execution environment, which is the focus of this paper. We base our works on OSGi service platform overlays, in the context of open service delivery chain as described in section 1.

3.2 Namespace Isolation

Core and Virtual Gateways. A core service gateway is a software element, managed by an operator. It makes resources available in order to run services. Such resources, physically supplied by the underlying hardware (*i.e.* the home gateway), include CPU cycles, memory, hard disk storage, network bandwidth, and optionnally standard services (*e.g.* logging, http connectivity). The gateway operator grants service providers access to these resources. This access is symbolized by a virtual service gateway, and provides a namespace isolation. Figure 5 illustrates this architecture.

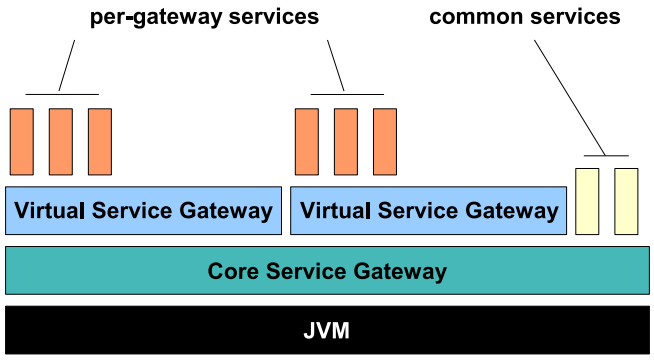

Fig. 5. Multi-service, namespace-isolated environment

Service cooperation. In an isolated, multi-application environment, applications are cloistered by default. However, they still should be able to cooperate on demand. In resource isolated environments, they cooperate through data communications. They either use standard OS facilities (*e.g.* sockets, IPCs, filesystem), or dedicated facilities (*e.g.* Links in the Java *isolates* API). By contrast, in open, non-isolated service environments, applications pass references on services (or interfaces). In an open, multi-service, multi-user, namespace-isolated environment, this still must be possible if explicitly permitted. The framework is then responsible for passing references.

3.3 Multi-user Java Environment

The gateway operator, through the core service gateway, acts much like a Unix root user. He allows users (service providers) to launch their shell or execution environment (their virtual service gateway). The core gateway also runs services accessible to all users. However, contrary to Unix root users, the core gateway does not have access to service gateways' data, files, *etc*, since these would belong to different, potentially competing companies. Figure 6 represents the architecture with participating users.

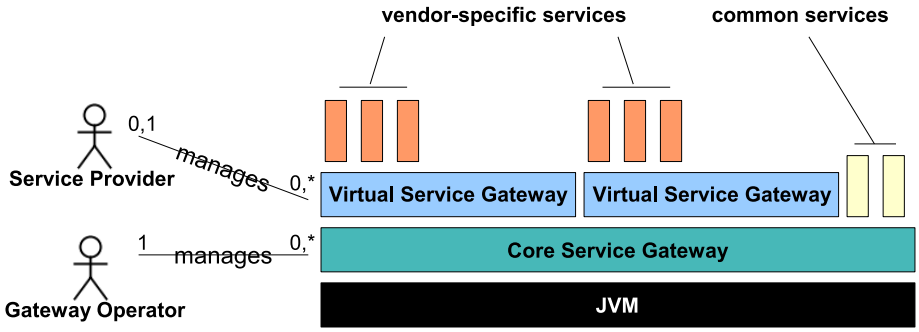

Fig. 6. Multi-service, multi-user residential gateway

The root user, *i.e.* the gateway operator, is responsible for the management of the virtual gateways it runs. Therefore, the core gateway must provide a management API, which is structured around 4 activities:

1. *Lifecycle management* provides a mean to start and stop virtual gateways;
2. *Performance management* provides information about the current status of a gateway (a virtual gateway or the core gateway itself);
3. *Security* is able to position credentials and make security challenges with core and virtual gateways;

4. *Accounting and Logging* brings information about service usage for each gateway.

Listing 1.1 is a code snippet for this management API. The functions use a parameter to identify the service provider's ID.

```
public interface VGWLifeCycleMgmt{
  public void    startVirtualGw(String providerId) throws Exception;
  public void    stopVirtualGw(String providerId)  throws Exception;
  public Vector  listVirtualGw();
  public boolean isAlive(String providerId);
}

public interface VGWPerformanceMgmt{
  public TimeTicks getCPUUsage(String providerId);
  public Memory    getMemoryUsage(String providerId);
  public Bandwidth getBandwidthUsage(String providerId);
}

public interface VGWSecurityMgmt{
  public void setCredential(String providerId, Credential newCred);
  public void challengeCredential(String providerId,
                                  Credential testCred);
}

public interface VGWAccountingMgmt{
  public String    notify(String providerId, Event evnt);
  public TimeTicks getServiceUsage(String providerId,
                                   String serviceClazz);
}
```

Listing 1.1. Core Gateway Management Interfaces

In addition, since the core service gateway can host services just as virtual service gateways can, it is also conform to the users management API described below (listing 1.2).

Users, or service providers, access their virtual gateways through a remote monitoring interface. According to the business model described in section 1, each service provider is responsible for the bundles she deploys. This means that service providers are encouraged to supervise their own services on their clients' service gateways. Also, some business services may inherently need remote monitoring: health care, home automation... Therefore, there are multiple reasons to define standard management interfaces for remote gateway control:

```
public interface BundleLifeCycleMgmt{
  public void    startBundle(String serviceId) throws Exception;
  public void    stopBundle(String serviceId)  throws Exception;
  public Vector  listBundles();
  public boolean isAlive(String serviceId);
}
```

Listing 1.2. Virtual Gateway Management Interfaces

4 Implementation

4.1 OSGi and Virtual OSGi

The service-oriented overlay we use for our prototype implementation is the OSGi Service Platform, an increasingly popular, industry-driven specification. In order to create virtual gateways, we propose a straightforward method. The core OSGi gateway runs several OSGi gateway instances (figure 5). It also instruments them and manages their life cycle.

As seen above, this service isolation architecture relies on an independency between the different actors around the gateway. The OSGi specifications define those actors as: (see figure 7)

- The *operator*, which controls the service platform in terms of connectivity and service deployment authorization;
- The *service deployment manager*, which acts on behalf of the operator and deploys new bundles.

The OSGi model considers one service platform, controlled by one and only one operator. Although the service platform server, which represents the hardware, can host many service platforms (each one ran by one operator), there is no relation between the operator and the service platform server.

In order to integrate the virtual service platform concept into this model, we need to adapt the OSGi reference architecture. An operator must be able to control a service platform server, in order to execute many virtual gateway instances. The curved arrow on figure 7 represents this amendment.

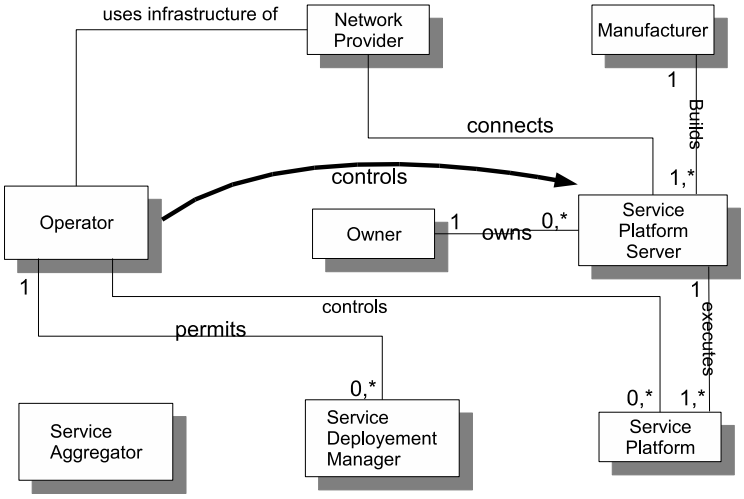

Fig. 7. OSGi Architecture Diagram

4.2 Service Isolation

The advantages of running several service gateway instances inside a single core gateway instance are quite immediate. Each service gateway can only access OSGi bundles and services it directly hosts. The core gateway itself does not see the hosted virtual gateways, but only a factory that allows their life cycle management. This is a straightforward way to enforce namespace isolation.

Each service provider sees his own virtual gateway as if it was a standard OSGi service platform. Therefore, at deployment time, standard OSGi deployment schemes come in action. At runtime, namespace isolation is provided through a hierarchy of classloaders. Each deployed component (*i.e.* OSGi bundle) comes with its own classloader, which delegates to its service gateway's classloader [15]. This way, by default, services from different providers (*i.e.* running in different virtual service gateways) are in different namespaces.

4.3 Service Cooperation

With proper class loading mechanisms, the core service gateway can provide services to its virtual service gateways. Currently, a static list of shared services and implementations is passed from the core gateway to virtual gateways. A more dynamic solution is planned, but not yet implemented.

4.4 Users and Management

Service providers use a remote monitoring console to manage push deployment and life cycle of OSGi bundles.

We developed a JMX infrastructure [16] that allows the core gateway to manage virtual gateways. It also provides the common management API described in listing 1.2, and allows to use per-bundle management GUIs.

4.5 Performance

We chose to run several OSGi Framework instances inside a core Framework instance. The main drawback to this approach is that it induces a resource consumption overhead. We ran a first set of performance tests on a standard Pentium IV PC, using a Gentoo Linux operating system, and Sun's JDK 1.5 with standard parameters (*e.g.* initial memory allocation pool). Our measures compare a vanilla Oscar 1.0.5 gateway with a core gateway that runs six virtual gateways, each launching a standard set of bundles. After 24 hours, we observe an overall 33% increase in memory use within the JVM's pre-allocated memory pool. This corresponds to a 2.9 MB consumption overhead.

4.6 Available Code

We provide several OSGi bundles that implement the concepts presented in this paper. They have been tested on Oscar [17] and Felix [18] open source implementations of the OSGi Service Platform specifications. These bundles are:[2]

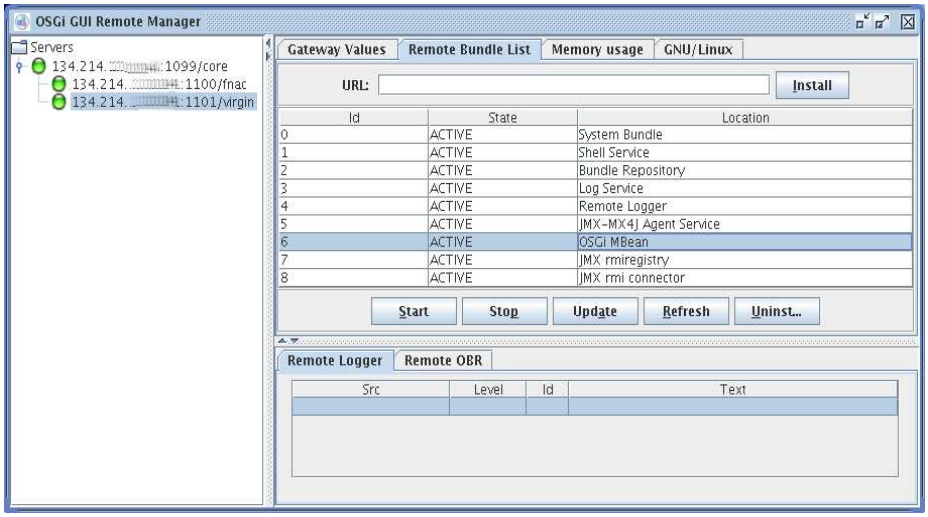

Fig. 8. Remote management console

- `vosgi.jar`: provides command-line tools to start and stop virtual gateways;
- `agent.jar`: embeds the main MBean registry;
- `rmiregistry.jar`: embeds a wrapper around the standard RMI registry;
- `rmiconnector.jar`: enables JMX remoting through JSR160;
- `httpconnector.jar`: enables the http management of the gateway.

5 Future Works

The works described in this paper assume that the same environment (the home gateway) is accessed, used and managed concurrently by several actors. Thus, there is a strong need for a security model. With contracts signed between service providers and the end-user, and between service providers and gateway providers, companies are liable for the correct execution of their services.

The only security mechanism we currently provide is namespace isolation. Other operations should come into play, such as:

- *Actor authentication.* Only the owner of a service gateway has the right to access and manage this gateway. An authentication scheme, for instance based on certificates, must be defined.
- *Signed deployment units.* Application deployment must be guaranteed. In particular, deployed components must be signed in order to prove their integrity.

[3] Available at `http://ares.inria.fr/~sfrenot/devel/` under the CeCILL open source licence.

- *Secure remote management.* Communications between a service gateway and its remote manager must also be guaranteed.
- *Secure local execution.* When running on a service gateway, services may be partially sandboxed. For instance, not every service should have the right to terminate the whole Java execution environment (`System.exit(0)`). This is somewhat similar to a partial resource isolation.

These mechanisms need to be further defined and integrated with our current architecture. This is the subject of ongoing works, and out of the scope of this paper.

6 Discussion and Conclusion

Our architecture is only a first step toward a multi-user, service-oriented Java virtual machine environment. These kind of virtual machines are a cornerstone for future remote, secured and efficient service management. This architecture can target the same markets as OSGi service platforms (Mobile telephony, vehicles, home gateways) and we consider it as an improvement of these infrastructures.

Since we use a standard virtual machine as the lower layer, we cannot have a real resource isolation control. We only rely on naming isolation as a first step. If we want to go further into resource isolation, we need to have an operating system that provides such a functionality. For instance, we should provide the same kind of multi-user view using real-time virtual machines. But these are not available on a large scale yet, and they are not aimed at this "in-the-middle" market (neither embedded nor high-end PCs).

This kind of multi-user oriented virtualization is beneficial for every actor in the service delivery value chain. If we look at the home gateway market as an example:

- For the service provider (virtual gateway): he is now able to provide his own services since he has his own semi-isolated environment.
- For the gateway operator (core gateway): he is free to provide his owns services. Gateway operators, who can be the same as the network access providers, can focus on IP-oriented services and leave other services to third party service providers.
- For the end-user: service gateways are becoming more and more autonomous and auto-managed. The sim plification of management should ease the acceptance of this kind of equipment by the general public.

The multi-user part of our proposal currently has two main alternatives. The first one is the multi-tasking virtual machine, and the second one is the use of "standard" operating systems (*e.g.* Linux, Windows).

From the MVM point of view, Sun's team works on mapping OS-level users rights within the JVM [19]. The resulting MVM-2 is aimed at big server systems that host many users and applications (SPARC/Solaris). Inversely, our approach

is focused on embedded systems, using standard JREs. Also, we believe that cooperation through service sharing, or interface sharing, is a better abstraction for developers than data sharing (`Link` objects between isolates).

Compared to standard operating systems, we assume that a service-oriented approach is a real improvement over "classical" C programming. We argue that both layers (Java and service orientation) are beneficial to service development for the targeted market. It enables many advantages (code structuration, code hot-plugging...) with only few drawbacks (execution time is one of the sole remaining).

References

1. Czajkowski, G., Daynès, L., Nystrom, N.: Code sharing among virtual machines. ECOOP (2002)
2. Czajkowski, G., Daynès, L.: Multitasking without comprimise: a virtual machine evolution. In: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2001) 125–138
3. Czajkowski, G.: Application Isolation in the Java Virtual Machine. OOPSLA (2000)
4. Czajkowski, G., Hahn, S., Skinner, G., Soper, P., Bryce, C.: A resource management interface for the java platform. Sun Technical Report TR-2003-124 (2003)
5. Golm, M., Felser, M., Wawersich, C., Kleinoeder, J.: The JX operating system. USENIX (2002) 45–58
6. Prangsma, E.: JNode. (http://jnode.sourceforge.net)
7. Hall, R.S.: A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks. In: Component Deployment: Second International Working Conference, CD 2004, Edinburgh, UK, May 20-21, 2004. (2004) LNCS 3083, pp. 81 - 96.
8. J. Corwin, D.F. Bacon, D.G., Murthy, C.: MJ: a Rational Module System for Java and its Applications. In: Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Language and Applications (OOPLSA). (2003) pp. 241-254.
9. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the ACM Symposium on Operating Systems Principles. (2003)
10. Yamasaki, I.: Increasing robustness by code instrumenting: Monitoring and managing computer resource usage on OSGi frameworks. OSGi World Congress (2005)
11. Whitaker, A., Shaw, M., Gribble, S.: Denali: Lightweight virtual machines for distributed and networked applications (2002)
12. Bieber, G., Carpenter, J.: Introduction to service oriented programming. http://www.openwings.org (2001)
13. The OSGi Alliance: OSGi Service Platform. http://www.osgi.org (2005)
14. Bieber, G., Carpenter, J.: Openwings, a service-oriented component architecture for self-forming, self-healing, network-centric systems. http://www.openwings.org (2001)
15. Liang, S., Bracha, G.: Dynamic class loading in the Java virtual machine. In: Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98). (1998) 36–44

16. Fleury, E., Frénot, S.: Building a JMX management interface inside OSGi. Technical report, Inria RR-5025 (2003)
17. ObjectWeb Consortium: Oscar OSGi framework implementation. (http://oscar.objectweb.org/)
18. Apache Software Foundation: Felix OSGi R4 Service Platform implementation. (http://incubator.apache.org/felix/)
19. Czajkowski, G., Daynès, L., Titzer, B.: A Multi-User Virtual Machine. In: Usenix. (2003) 85–98