Java OpenBinder Extension

Radu-Corneliu Marin
School of Computer Science and Automatic Control
Politehnica University
Bucharest, Romania
Email: radu.marin@cti.pub.ro

Daniel Urda
School of Computer Science and Automatic Control
Politehnica University
Bucharest, Romania
Email: daniel.urda@cti.pub.ro

Nicolae-Vladimir Bozdog
School of Computer Science and Automatic Control
Politehnica University
Bucharest, Romania
Email: nicolae.bozdog@cti.pub.ro

***Abstract*—Although Java is renowned for its myriad feature set, we found that it lacks in performant Inter-Process Communication. A somewhat recent development in said domain, the OpenBinder C/C++ framework, has introduced a novel inter-process communication abstraction which we believe could be easily extended into Java by using the Java Native Interface. In our paper, we introduce JOBe, namely the Java OpenBinder extension. Such an endeavor has been attempted a few years back, when the Google team that developed Android designed and implemented the core of the aforementioned software stack: the Binder framework. Based on their concepts, we are extending the OpenBinder framework into Java as to offer extremly flexible Remote Method Invocation, based on the Binder abstraction.**

I. Introduction

As Java is growing in popularity, its flaws are becoming more apparent each day. One of the most pressing limitations, it being the one we are interested in, is the lack of performant IPC - inter-process communication - methods.

A rudimentary technique in Java for communication between processes (and even for programs written in different programming languages) is using a protocol over sockets in order to broker correspondence between two ore more entities running in the same process or in other processes, on the same machine or on remote machines. Such a method is cumbersome to use, it implies a huge amount of effort to design, to implement, to maintain and to debug secure inter-process communication. We consider such a method as being obsolete and do not consider it to be valid.

For years now, many frameworks and standards have been developed for RPC - remote process calls - or for distributed objects, such as XML-RPC, respectively CORBA, and most of them have received praise, though none managed to become a de facto standard in said domain. Although, not one of these solutions were designed with respect to inter-process communication, they represent valid choices in this direction. Because these solutions present penalties, usually represented by large time to execute the IPC call, we are not interested in them.

Probably one of the most notable solutions for IPC in Java is RMI - Remote Method Invocation - which "enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines*, possibly on different hosts" [4]. RMI, being itself a Java techonology, is easily integrated into basic Java code, but because it was designed and implemented for transparent remote calls on machines over a network, it has a consistent overhead in dealing with an IPC call.

In our paper, we present a novel inter-process communication method based on a native C++ solution, namely OpenBinder. The basic difference between this solution and the previously mentioned solutions, is that OpenBinder is strictly oriented for IPC, as opposed to the latter. Instead of using sockets for correspondence between processes, OpenBinder has introduced a unique method: passing a method call to a Linux kernel module, which, in turn, reifies the identity of the caller and of the callee and forwards the call to the remote process (and in fact returns the result back to the caller). Therefore, porting such a solution into Java is not compatible with most operating systems, other than some Linux distributions. This issue is of no concern for us, as OpenBinder can be ported onto other platforms as well, while the Java interfaces remain untouched.

Our paper is structured as following: Section 2 presents the background for our work, most importantly it describes OpenBinder and Android's Java extension for it. In Section 3, we present the design and architecture of our solution, whilst Section 4 entails its implementation and issue coverage. We end our work with our observations, conclusions and plans for forthcomming features in Section 5, respectively Section 6.

II. Related work

Although OpenBinder introduces numerous interesting concepts, it has not yet been popularized or documented. This is mostly due to the fact that the OpenBinder project was abandoned about the time that Android's Binder was being designed and implemented. Therefore, we feel obliged to render a detailed description of the inner-workings of OpenBinder and to provide an overview of Android's view over the Binder mechanism. This second section is dedicated to the aforementioned subjects.

A. OpenBinder

Dianne Hackborn, former manager of the OpenBinder project and famous engineer at BeOS, PalmOS and currently Google, described OpenBinder as "a new approach to operating system design, in the same vein as a lot of other attempts

at introducing a modern object oriented approach to operating system design" [2].

The OpenBinder introduces an interesting abstraction, namely the Binder, which is used to design and implement a distributed component architecture by means of abstract interfaces, their behaviour and their implementation: "As long as you are using these interfaces, the actual object behind them can be written in any language with Binder bindings, and exist anywhere your own address space, in another process, or even (eventually) on another machine" [3].

In broad strokes, OpenBinder is similar to CORBA on Unix, but unlike most distributed component architectures, the Binder is "system oriented rather than application oriented" [3], in other words it is designed to support system-level components having its focus not on cross-network communication, but on inter-process communication.

The core of the framework is the *smooved* executable which is the main Binder server. Its main purpose is to set up the Binder environment by creating a Root Binder Context by dint of which components (such as services) will be made available. The *smooved* executable is also responsible for starting the **Package Manager**, which will host the components, and for starting the **Process Manager**, which will run the components in.

The Binder abstraction alone is used only for intra-process communication and, in order to take advantage of the multi-process bindings, the OpenBinder kernel module needs to be built and inserted into the kernel. The module's main purpose is to create and manage a thread pool which takes care of the inter-process bindings, also called a root namespace. It is designed as a character device placed in **/dev/binder** and it is interfaced by means of *ioctl* calls. The Binder module is responsible for reifying the identity of the calling process and the identity of the callee process. It marshals the parameters sent for the remote method invocation and the results returned after execution through the kernel from the host process to the remote process and back. The custom kernel module is used instead of standard Linux IPC facilities in order to model IPC operations as thread migration. Although this component is optional (all other OpenBinder features will function properly without it), it is the backbone of the inter-process communication solution.

The reification process in the kernel module is done by means of file descriptors: if a user-space thread participates in Binder IPC, it has to open the **/dev/binder** and the file descriptor obtained is used to identity the initiator and recipient of Binder IPCs. The IPC mechanism makes use of the file descriptor in order to interface with the driver through a small set of *ioctl* commands:

1) **BINDER_WRITE_READ** - used for dispatching Binder commands after which blocks to receive incoming operations and related results. This is the key command for all Binder IPC.
2) **BINDER_SET_WAKEUP_TIME** - schedules a user-space event at a given time.
3) **BINDER_SET_IDLE_TIMEOUT** - sets timeout for threads awaiting Binder operations.
4) **BINDER_SET_REPLY_TIMEOUT** - sets timeout for awaiting replies for Binder commands.
5) **BINDER_SET_MAX_THREADS** - set the number of threads in the thread pool.

A Binder IPC operation is also called a **Transaction**.

Last, but not least, the kernel module has another key responsability in the communication mechanism: it performs mapping of objects from one process to another. An object reference has two mutually exclusive forms: an address - local references - and a handle - remote references. The local process sees a Binder reference as an address in its own address space, but when initiating a Transaction it sends a 32bit handle to that reference, which the remote process sees as a point to the object in its local address space. The module performs the translation between pointers and handlers, being the sole mapping manager.

That having been said, if the Binder module is inserted into the kernel, the *smooved* executable has the responsability of becoming the global host of the root namespace created by the kernel module, in other words *smooved* assumes the role of host of the root **SContext** namespace. Each Binder is passed in a reference of **Context** representing the link to the OpenBinder environment. For situations in which Binders are not present, references to **Context** can be obtained from the *smooved* process by means of the **SContext** class as following:

1) `SContext SystemContext()`
   This method is a back-door to obtaining the root or system context. Although it is initially created by *smooved* with full permissions, access to it is restricted and may not return a valid context.
2) `SContext UserContext()`
   This method is a back-door to obtaining the global user context. This returns the least-trusted context which is always guaranteed to exist, but the functionality offered by it is fairly limited.

OpenBinder also offers a **Support Kit** consisting of a suite of classes and utilities which ease the development of Binders. The most significant components are the following:

1) `SValue`
   The SValue is a generic container for a blob of bytes that has a type code associated with it. Its most significant feature is its genericity: it can contain complex data structures or simple typed data.
2) `SParcel`
   The SParcel is a container of raw block data and it is used to marshall SValues for inter-process communication.
3) `SAtom`
   The SAtom is a base class for reference counted objects. By definition, Binder objects are reference counted (in order for the kernel module to take them into consideration). Classes should either inherit from it and all pointers to an SAtom must use a sptr<T> and wptr<T>

smart pointer template classes instead of raw pointers.

OpenBinder uses IDL - Interface Definition Language - to ease generating Binder interfaces. The framework provides the *pidgen* tool which parses IDL interfaces and generates necessary code for Binder transactions. IDL interfaces may import other interfaces and can define functions - Binder methods, events - methods called on specific events - and properties - Binder state values. The *pidgen* tool mainly generates two classes both of which implement the defined interface and that derive from the **BBinder** class - the standard implementation of the core Binder interface, namely the **IBinder**:

1) a **Binder** class - the class which will implement the interface and which will be registered in the **Package Manager** by means of the Binder root SContext.
2) a **BinderProxy** class - the class which provides a skeleton for remote accessing the Binder class instances. It provides stubs for all the methods in the interface; each stub marshalls its parameters, initiates a Binder Transaction, waits for results and returns them in a transparent fashion.

The only issue that OpenBinder raises is its *old age*. Due to the fact that the project was abandoned in 2005, it has become deprecated: the main problem is in the kernel module that has been developed for the 2.6.10 Linux kernel and porting it to the latest Linux kernel is a complicated task and not part of the current article.

B. Android Binder

Considering that "the Binder could be described as a <<framework framework>>. It doesn't do anything itself, but is an enabling tool for implementing other rich frameworks, such as the view hierarchy, media framework, etc" [3], Android has taken it literally and has built all the System Servers based on the Binder framework. In order for this solution to be feasible, they have enriched the OpenBinder kernel module and have ported it to a newer Linux kernel version, namely 2.6.35.

The novelty of their solution is that they did not just bring the OpenBinder framework up to date, but they have also extended the native C++ implementation into Java as well. By so doing, they were able to create a safe, fast and dependable Java based software stack based on the Binder IPC Model.

Although the Android Binder infrastructure is based on the original OpenBinder implementation, there are also differences:

- The smooved server is no longer used. Instead Android defines the ServiceManager Java class (which also has native C++ hooks as well), which is responsible for registering all Binders for System Services such as the Sensor Manager, the Battery Manager, the Activity Manager and many more.
- Android has abandoned the pidgen tool. Instead it provides the aidl tool which no longer offers support for native C++ IDL files, but for Java IDL interfaces (which are modelled like standard Java interfaces).

Android has also extended the functionality offered by OpenBinder with the following features:

- File descriptors may be passed between processes,
- Memory (heap regions) may be shared between processes,
- Binders may be transacted between processes.

A more accurate overview of the Android Binder Java extension can be seen in Figure 1. As can be observed the Java layer sits on top, mapping the Java IBinder onto the native one. The Java Binder is mapped onto the native Binder by means of helper classes such as JavaBBinderHolder and JavaBBinder which are responsible for converting the instances between Java and C++.

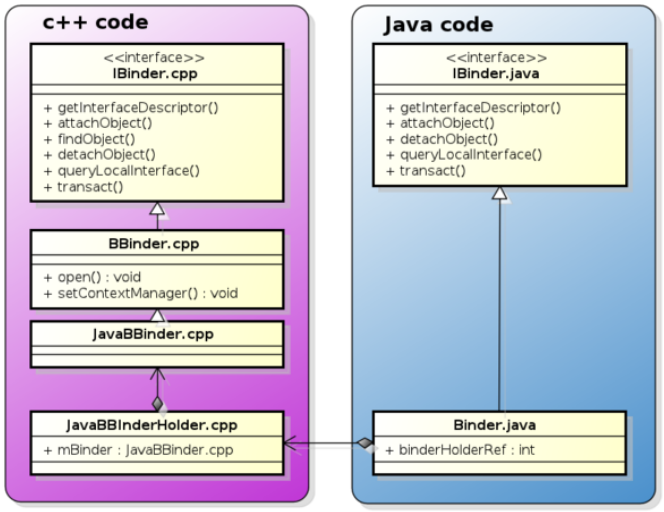

Fig. 1. An Overview of the Java binder implementation [1]

The core of the Android Binder framework is the IBinder interface which is implemented by all Binder components, NativeBinders and BinderProxies alike. The IBinder defines a crucial method, namely **transact()** which, on the client side marshalls the data and method code to the Binder kernel module and waits for results, and on the server side it invokes a Java callback **onTransact()**, marshalls the results and sends them back to the kernel module. Marshalling is performed by means of the Parcel class which serializes primitive types such as int or float, but also live IBinder objects and serialized Parcelable objects.

We consider the Android implementation as our base state-of-the-art reference and its influences are noticeable in the following sections, namely Architecture and Implementation.

III. Architecture

The third section describes the JOBe proposed architecture which contains elements from both OpenBinder and the Android Binder, but also components introduced by us.

Figure 2 illustrates the JOBe architecture. The following components are directly visible or indirectly deductible from datapaths:

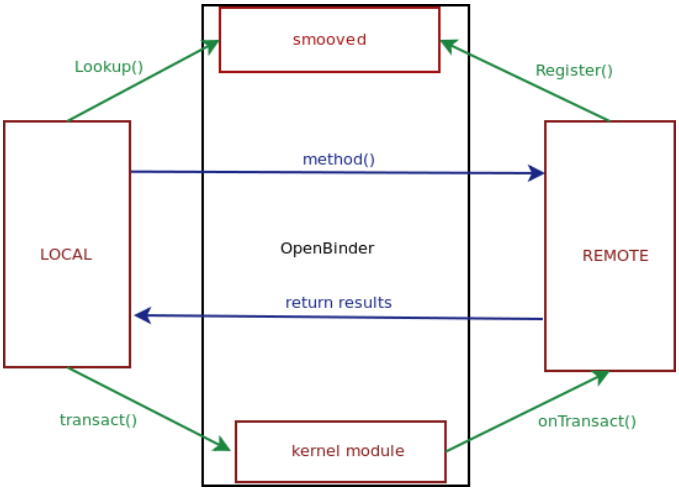

Fig. 2. JOBe architecture

- **OpenBinder core**: in order to be easily compatible with Linux distributions, we have preserved the OpenBinder smooved server and kernel module. These components offer the means to register/lookup binder services (instead of implementing yet another server which has to be set up) and also provide the functionality that allow kernel Binder calls. The Binder is continuosly running in the Linux system, in order to ensure the availability of the inter-process communication mechanisms.
- **JOBService**: it is not visible in the Figure 2, but the green datapath offers insight into its functionality. It is an OpenBinder native service which actually represents the communication media by means of which Java Binder calls will run upon. The simplicity of this service is given by our solution in which the native service does not need to know about the actual Java method it is executing, it only acts as the communication channel by which Java messages are passed from process to process, from virtual machine to virtual machine.
- an **interface** between the Local and Remote objects represented by means of the blue datapath. This is an implicit component describing the agreement between local and remote objects. Java interfaces need to be parsed by an automatic tool which offers a formalization for Java method calls. In this sense, our framework extends the definition of a regular Java method invocation, that of being a standardized method to send a message to an Object. Each method is standardized as a tuple <**Code**, **Arguments**, **ReturnType**>, where **Code** is a unique code identifying the method in the interface, **Arguments** being a container describing the argument data with their associated datatypes and **ReturnType** being the datatype associated with the method result.
- **Local Object**: will be further referenced as Binder Proxy. This component represents the local connection to the remote object. It contains stub implementations of the interface which marshall arguments, initiate a Binder transaction by means of the JOBService and wait for results or Exceptions.
- **Remote Object**: will be further referenced as Binder. This component actually offers the true implementation of the interface. It (probably) lives in a separate process and is accessed by means of the JOBService. It unmarshalls the arguments, executes the methods body and marshalls return values or Exceptions back to the Binder Proxy through the JOBService.
- **Parcel**: means of marshalling data. All Binder methods pass arguments and returned values through serialized data containers, namely Parcels. A Parcel contains raw data with associated datatype. The Parcel class has equivalents both in Java and C++ and a sound mapping between the two leads to more feasible marshalling and unmarshalling.

As can be observed from the detailed architecture, the Android Binder has left its mark on our solution due to the fact that we could not ignore our predecessors in this direction. Although, the Android solution is complete and sufficient we believe that our changes infer different, but valuable, semantic information: we have chose not to add the IBinder Java interface, which in Android was implemented by Binders and Binder Proxies alike to show the relationship between the two components, even if they both implemented the cross-process interface as well. We believe that in a semantic sense our solution separates the Binder abstractions from the interface itself, as to illustrate a more transparent system to the programmer. In order to reflect this, we differentiate the local and remote object by separate components the Binder and the Binder Proxy in which the only common point between them is the cross-process interface they are implementing. This solution is also complete and sufficient for our needs, because, as opposed to Android, we do not plan on sending Binder objects by means of Binder IPC calls.

The true transparency provided by our framework is illustrated in Figure 2 as the blue datapath. From the programmer's point of view, all the aforementioned components are not visible because the Binder inter-process method invocation appears to act as regular Java method invocation. The only apparent difference is that Binder method invocations must treat the case of remote exceptions.

IV. Implementation

This section covers the actual implementation of the architecture presented in the previous section. In designing and developing the JOBe framework we have used the Java and C++ programming languages, the OpenBinder framework and, in order to create the links between them, we have used the Java Native Interface. In the following subsections, we have covered all of our tools and components in detail.

A. JOBService

As mentioned in the previous section, the JOBService is a simple OpenBinder component which is responsible for acting as the communication media by which Java callbacks are

passed between processes. In implementing said component we have used the IDL description language provided by OpenBinder. The interface is depicted below:

```
interface IJobService {
    methods:
    void onTransact(  SParcel in,
                      SParcel *out,
                      int methodCode);
}
```

The purpose of this interface is to define an OpenBinder service which serializes arguments through the in SParcel and return values through the out SParcel, and passes both alongside the unique methodCode to identify the Java callback that needs to execute. Using the pidgen tool, we have obtained the BnJOBService and BpJOBService, representing the native Binder, respectively the native BinderProxy. The BnJOBService is responsible for mapping the in SParcel to a Java Parcel and passing it along with the methodCode to a Java callback registered when creating an instance of the native Binder; also the Binder waits for the Java code to execute, marshalls the results into an SParcel and ends the Binder call. The BpJOBService is the mirrored component, which actually maps the arguments Parcel into an SParcel, initiates an OpenBinder transaction sending the arguments alongside the methodCode and waits for the transaction to return the values, which are marshalled and sent back into the Java code.

This component is built only once, and used by all Java Binder classes. In this sense, we have introduced a naming scheme in order for the smooved server to distinguish between different components: the Java interface name and package are passed into the BnJOBService and BpJOBService and their concatenation is used to register or lookup the component in the OpenBinder framework.

B. Binder

The Java Binder is defined as an abstract class which maps onto the BnJOBService. It is declared as following:

```
abstract class Binder {
    public abstract void
    onTransact(  Parcel in,
                 Parcel out,
                 int methodCode);

    public abstract String
    getInterfaceName();

    private native void
    initialize();

    static {
        initialize();
    }
```

The native initializer is used to create a BnJOBService instance, pass it a reference to the onTransact abstract method and the yet unknown interface name (represented by the abstract getInterfaceName method). The class initializing is done in a static context, as to guarantee that when the implementing class is loaded, the BnJOBService tied to it is registered once and only once to the smooved server.

An outstanding issue that occured in the implementation of this component is its registration to the smooved server. As mentioned in the second section, in order for a service to be taken into account by the OpenBinder framework it needs to obtain an external SContext instance. Although, the OpenBinder implementation offers two Binder contexts: user and system, neither of them is able to register our component into the system - the user context does not have enough privileges and the system context returned by the framework is not valid. The solution to this problem is to re-implement the Package Manager included in the smooved server and such an endeavor is not included in the current paper.

C. Binder Proxy

The Java Binder Proxy is also an abstract class which maps onto the BpJOBService. It's implementiation is presented below:

```
abstract class BinderProxy {
    private native void
    transact(  Parcel in,
               Parcel out,
               int methodCode);

    public abstract String
    getInterfaceName();
    }
```

In comparison with the Binder class, the BinderProxy is much simpler: it also complies to the naming scheme, fact reflected by the abstract getInterfaceName() method and what it actually does is forward all calls to the native transact() method. In C++ code, this looks up the service on the smooved server and initiates a Binder transaction passing in the arguments passed from Java code.

As mentioned in the Binder implementation, the BinderProxy is also affected by the SContext issue as it cannot lookup a service that could not be registered to OpenBinder.

D. Parcel and Parcelable

The Parcel concept is borrowed from the Android solution. It is a container for raw data and it also contains datatypes associated with the data within. Our Parcel class is currently implemented using a Java Vector which retains data and type as well. We provide methods for marshalling and unmarshalling primitive types. In order for Objects to be passed between virtual machines we have implemented the Parcelable interface, also borrowed from the Android Binder. It's purpose is to force serialization:

```
public interface Parcelable {
    Parcel toParcel();
}
```

Our Parcel model maps upon the SParcel class defined by the OpenBinder SParcel datatype which may be passed in native Binder transactions. Currently our mapping is probably not the best, because conversion between the Java objects and the C++ instances cannot be automatized. This creates an issue for the JIDL tool presented in the following subsection. A possible fix for this problem could imply passing the address of a SParcel pointer to the Java instance as well. By so doing, all operations would operate directly onto it and conversion would be a matter of accessing an internal private field. Unfortunately we have not been able to implement it yet.

E. JIDL tool

The JIDL tool is in many ways similar to pidgen. It takes an interface and automatically generates code comprising all interactions with Binder internals. By so doing, it eases the programmers burden to understand the entire framework before actually programming and also reduces the time of development for applications. The JIDL tool has one advantage over the native OpenBinder tool: it used Java regular interfaces to define the agreement between Binders and Binder Proxies. By so doing, it takes advantage of the Reflecion API provided by Java for retrieving method information and it also gains all of the time wasted by pidgen to check the validity of the IDL interface.

Being provided with a valid Java interface, JIDL's functionality can be sinthesized into the following steps:

- for each method generate a unique integer identifier and based on its signature generate the marshalling and unmarshalling code
- generate the Binder class - it is actually an abstract class as it does not offer implementation for the interface, it is the programmer's duty to provide such information after the code generation phase ends. The most important feature generated is the onTransact()'s body: it groups the unmarshalling code previously generated by unique identifier for each method and invokes the cross-process interface methods (which will be implemented by the programmer). This ensures the closed loop, as the onTransact() method is invoked from native code when receiving a Binder transaction.
- generate the BinderProxy class. This is no longer an abstract class as it provides stub implementations for all methods. For each method, it uses the marshalling code generated above to serialize the arguments and calls into the native C++ code in order for the BpJOBService to initiate the Binder transaction with the appropriate component. The results returned by the Binder transaction are passed back into the Java onTransact() callback.

After the code generation is finalized, the programmer only needs to implement the Binder's code for the cross-process interface.

As can be seen and how we actually mentioned in the "Architecture" section, the entire development process is transparent to the programmer as he or she is oblivious of the inner workings of our framework.

V. Experimental Results

The most significant outcome of our project was building and running the OpenBinder framework. Due to the fact that the project was abandoned more than 7 years ago, the build process was tedious and cumbersome and required most specific features: the 2.6.10 kernel and headers, gcc 3.4.0, bison 1.875d, flex 2.5.31 and Java 1.5. This prooved to be the winning configuration that permitted us not only to build OpenBinder but also applications based on the aforementioned framework, such as the Binder Shell. Based on the Binder Shell, which emulates a system shell and runs on top of the smooved server, we have managed to manually register Binder components.

Unfortunately, because of the SContext issue presented in the previous section, we could not test our framework. Allthough we could not present empirical data, we consider it to be common sense that the Java Binder framework offers most feasible inter-process communication because instead of using sockets which are renowned for high latency as most IPC solutions do, we implement our mechanism by calling into a Linux kernel module which further pushes our transactions to the remote process.

VI. Future work

We consider the OpenBinder framework as having the highest priority, it being the core of our framework. In this sense, not only do we plan on bringing the infrastructure up-to-date with the current Linux kernel, but we would also port it to other platforms as well, such as Windows or Mac OS X. But also considering our encountered issues we feel it would be best to first re-implement the Package Manager from the smooved server as to provide means to register OpenBinder components from external contexts. By so doing, we would offer powerful justification for porting onto other platforms by means of a proof-of-concept implementation on Linux systems.

VII. Conclusions

In conclusion, we sustain our initial motivation: Java needs powerful and feasible inter-process communcation mechanisms. As Android has prooved with their stunning success,

the OpenBinder IPC model seems to be the most suitable solution for modern operating systems and we strongly feel that extending it into Java will have a major impact on the Java application development.

REFERENCES

[1] Kees Jongenburger. Android binder, 2011.
[2] Eugenia Loli-Queru. Introduction to openbinder and interview with dianne hackborn, January 2006.
[3] OpenBinder. Openbinder, 2005.
[4] Oracle. Remote method invocation home, 2012.