ROCS: a Remotely Provisioned OSGi Framework for Ambient Systems

Stéphane Frénot*, Noha Ibrahim†, Frédéric Le Mouël*, Amira Ben Hamida*, Julien Ponge*,
Mathieu Chantrel*, Denis Beras*
*Université de Lyon, INRIA Email: firstname.lastname@insa-lyon.fr
†Grenoble Informatics Laboratory, Grenoble, France firstname.lastname@imag.fr

***Abstract*—One of the challenges of ambient systems lies in providing all the available services of the environment to the ambient devices, even if they do not physically host those services. Although this challenge has come to find a solution through cloud computing, there are still few devices and operating systems that enable applications execution by only uploading the required components into the runtime environment.**

The ROCS (Remote OSGi Caching Service) framework is a novel proposal which relies on a heavy-weighted standard Java/OSGi stack. It is distributed between class servers and ambient devices to provide full functionalities to resource-constrained environments. The ROCS framework provides improvements in two areas. First, it defines a minimal bootstrap environment that runs a standard Java/OSGi stack. Secondly, it provides an architecture for loading any necessary missing class from remote servers into memory at runtime.

Our first results show similar performances when classes are either remotely downloaded into the main memory from a local network or from a flash drive. These results suggest a way to design minimalistic middleware that dynamically obtain their applications from the network as a first step towards cloud-aware operating systems.

I. Introduction

In ambient environments, equipment units take into account their local context to run applications adapted to their constraints. These applications are rather ephemeral and can be updated / uninstalled on a regular basis. We call ambient devices small devices that host these applications and integrate an operating system that enables this dynamic behavior.

Virtual machines are becoming a standard for deploying and loading components required to execute an application. Such applications are composed by assembling components. This approach allows the virtual machine to easily identify and isolate the classes to load. We assume that a "good" operating system may not have to download the components but rather only their descriptions. When needed, the operating system provides the necessary component resources and executable instructions directly from the network into the device memory. Our concerns are emphasized by our xDSL-Box targeted equipment, where a box is an intermediate equipment that lays between the end-user applications and the network provider DSLAM. In this model, boxes are physically deployed at the user home and have no extension facilities (e.g., it is difficult to increase memory or hard-drive capacities). In ambient environment where connectivity is (mostly) always available, it is worth adopting this approach for dynamically loading classes without physically storing them on the box. This approach, which uses components that are not already available from the devices, is very much related to cloud computing. Briefly, cloud computing [1] is about services being encapsulated, having APIs for accessing them, and being always available over the network. This makes for a relatively straightforward parallel with our cloud operating system.

Among the possible execution environments for ambient environment, Java/OSGi [2] holds an interesting position. Indeed, it allows an easy deployment of applications, in the form of jar files called *bundles*. OSGi also takes advantage of the type safety of Java, and enhances the programming model by relying on a service-oriented approach. We believe that OSGi is the most promising solution for dynamic systems dedicated to ambient environments. Yet, we believe that OSGi has one major drawback: implementations download new applications executable code and resources into a local cache. In our opinion, this behavior is the main hindrance to the integration of the OSGi platform in ambient environments. Caches contain all the resources that a given application may need, even though a significant proportion of classes is often never used throughout executions, which drastically reduces the number of applications that can be installed as space is wasted.

This article presents *ROCS*, an elegant solution that relies on the OSGi components approach for application life-cycle management, and on Java remote class loading mechanisms for deporting classes deployment to a remote server. With ROCS, OSGi application resources are never stored on the ambient device since they are downloaded only when strictly necessary directly into the Java Virtual Machine memory. Nevertheless, in order to host a standard Java SE/OSGi runtime stack we also have to find a solution for running a full class 40MB Java/OSGi system on a 8MB flash environment. The ROCS framework enables a *minimal bootstrap environment* for the Java/OSGi stack that fits within the 8MB limitation. This first result shows that we can have new home equipment with limited storage capacities only for caching component description. Another result shows that the network downloading of classes is mostly equivalent to getting them from a slow flash hard-drive, and by way of consequence is viable in a mostly "always-connected world".

Section II presents the necessary technical context to understand the ROCS architecture which is detailed in Section III.

Section IV presents test results that confirm a real interest in using our approach, as performances are not downgraded compared to a local execution runtime. Finally, Section V concludes the present article.

II. Technical Context

The following section starts by providing a review of start of the art that relates to ROCS. Then, we briefly recall the concepts and main technical aspects that are useful for understanding this paper concerning Java classes loading and the OSGi framework.

A. State of the Art

Current ambient operating systems such as Android[5], iPhone OS [3] or J2ME [4] rely on virtual machines and component oriented approaches. Programming lightweight embedded systems has become the problem studied in many research works [5]. They focus on developing lightweight components and services in order to deploy them on resource-constraint devices. These architectures adopt the same approach: when a customer needs an application (s)he downloads it and starts it from the local device. All resources, whether provided or needed by the application, are locally provisioned (e.g., downloading an application from the Apple AppStore). By contrast, ROCS defends an approach that initially downloads each component description for controlling local compatibility, and then only fetches the required resources into memory, without ever storing them locally. The ROCS concepts are totally agnostics of service frameworks such as OSGi, Android or J2ME, as it relies on low-layer virtual machine capabilities. Related works focus on runtime optimizations, whereas we focus on provisioning optimization. This provisioning issue is directly related to cloud computing [1] where resources are available from somewhere on the network and are being delivered on demand.

Many embedded Java solutions rely on specific tailored implementations [6], [7], [8], [9], and [10]. Given that the standard Java runtime class library is too large to fit on ambient devices local disks, most operating systems do not use Java SE and focus on dedicated, stripped-down libraries for these environments.

Still, existing technologies such as Java RMI and Java Applet enable to remotely and dynamically install new classes at runtime from remote locations. Actually, the *hidden* standard Java mechanism relies on class loaders in the virtual machine to define classes provided through remote locations.

Finally, OSGi relies on a bundle approach that isolates a subset of classes that take part in the realization of an application. Isolation is provided through a specific class loader infrastructure which deviates from the "standard", tree-based way of organizing classloaders into hierarchies.

Our architecture is based on the tools provided through standard Java remote class loading architecture, and on OSGi specification to isolate and define applications needs. In the remainder of this section, standard Java class loaders structure as well as some elements of the OSGi architecture are presented.

B. Java Class Loading

Java classes are defined in files containing their corresponding compiled Java bytecode (`.class` files). Classes are loaded on-demand by specific Java object instances of `ClassLoader` subclasses [11]. Class loaders must find classes and must store their bytecodes in memory. Class loader objects are usually triggered when a constructor invocation is performed for the first time on a given class. For instance, the first time the line `"new java.util.Vector()"` is encountered the class loading architecture loads the related class bytecode in memory, thus effectively loading the class definition, and invokes the constructor method. There are three standard class loaders. First, the *bootstrap* class loader handles the classes that are part of the standard Java runtime, stored in a file which is usually `rt.jar` on Sun-derived implementations. Secondly, the *extension* class loader handles classes which are in a specific location of the Java runtime environment (e.g., vendor-specific implementations of the cryptographic libraries). Finally, the *system* class loader handles user-defined classes made available on the file system through the `CLASSPATH` environment variable. If Java class loading was limited to these three class loaders, Java would not be so dynamic, as it could not automatically retrieve classes at runtime from dynamically defined locations. Given that class loaders are standard classes, they can be extended to deliver customized functionalities. One classical behavior consists in retrieving classes from remote locations through the `URLClassLoader` class loader, which points to remote URLs where new classes definitions and resources can be found.

The classes loaded by a given class loader are isolated from each other. In particular, they are executed in a dedicated namespace [12], which prevents naming conflicts and uncontrolled access between classes developed and provided by independent vendors.

Class loading obeys the following rules:

1) The caller class loader is used. For instance if code in class `A` makes a `new B()` call, the same class loader that was used for loading `A` is used to load `B`.
2) When a class loader loads a class, it must first ask its parent class loader wether it already has the definition or not. It means that local classes are loaded before remote one and system classes are loaded first. This approach avoids overriding standard classes, it improves isolation, it enables class versioning, and finally, it improves security.
3) When a class loader loads a class, all classes belonging to the same package must be loaded by the same class loader.
4) Some weird restrictions are put on `java.*` and `javax.*` packages. Some implementations impose that `java.*` / `javax.*` classes have to be loaded by the bootstrap class loader. The historical reason is that the *bootstrap* class loader is "trusted".

If a class loader conforms to these rules, then it can load classes from any place and it can isolate new classes definitions from each other. This approach of a dedicated class loader to load a complete application is the cornerstone of the OSGi framework.

C. OSGi: a multi Java applications framework

The OSGi specification defines a multi application layer on top of Java. The main idea is to package applications as collections of classes and resources in Java archives handled by an OSGi framework. Each Java archive is then called a *bundle*, which is typically downloaded from a remote location and installed locally before it can be *started*. When it is started, the OSGi framework associates the bundle with a *module loader*, which is a specific class loader dedicated to the newly installed application. The module loader isolates the resources contained in the bundle from other bundles. Then, the OSGi framework instantiates a class from the bundle, which is implements a specific Java interface (`BundleActivator`), and which forces the class to implement two methods that are called on two lifecycle events: *start* and *stop*. This start/stop sequence enables the materialization of a multi-application layer above Java. A bundle usually contains:

- a descriptor file (Java manifest format),
- a startup class that implements the `BundleActivator` interface,
- Java classes from one or several Java packages,
- resource files, such as pictures and other data,
- embedded archives (JAR files within a JAR file), and
- native libraries.

One key feature of the OSGi framework is the seamless support of application deployment: new applications can be installed, updated and uninstalled at runtime without requiring a restart of the Java virtual machine, as the large majority of non-OSGi applications do.

The OSGi specifications state that bundles must be saved, along with their state, so that an entire service platform can be stopped and restarted without any loss of configuration information. In order to achieve this, OSGi implementations use a cache on the local file system. Most implementations associate a cache with a named user profile, or profile ID, in a directory such as `.<osgiImpl>/<profileName>/` for convenience purposes. Each profile contains its own set of installed bundles.

The cache structure stores all bundles that have been installed during the running framework. Each time a bundle is started from a remote URL, the bundle is first downloaded from the remote location, then unpacked and renamed in the local cache, before being finally started.

The next section describes the ROCS architecture that creates a shadow cache of locally installed bundles and defines a minimal bootstrap environment for the OSGi/Java stack.

III. A Remotely Provisioned OSGi Framework

In this section, we first describe the general ROCS architecture, then we detail the ROCS server specific architecture. Finally, we present how we designed the minimal runtime environment for bootstrapping OSGi/Java stack.

A. General Architecture

The ROCS framework relies on two elements: *Ambient* devices that run standard OSGi frameworks, and *Remote Cache Servers* that deliver class bytecode on-demand. Our architecture is 100% compliant with the OSGi specifications. The only difference is that bundle-embedded classes and resources are initially not downloaded onto the ambient devices.

The ROCS server downloads each requested bundle from its remote bundle repository and stores it locally. Then, the ambient device framework downloads the meta-data contained in the bundle manifest. Once the attributes have been obtained from the ROCS server, a local shadow cache is created. It has the same structure as the standard implementations, except that resources are not deployed locally.

When this deployment phase ends, the bundle is into the *Resolved* or the *Installed* OSGi status. If the bundle is *Resolved*, it can be started. The starting process instantiates the class whose name is identified by the *BundleActivator manifest property* and invokes its *start* method. Thus, when the *BundleActivator* class needs to be created, its class bytecode is downloaded from the ROCS server and brought straight into the memory of the ambient device through remote calls. If the `start` method does not throw any exception, the bundle turns into the *Started* OSGi status. Once it has been started, every time a new class or resource from the bundle is needed, a new call is made to download the corresponding bytecode as an array of bytes[1] from the ROCS server.

Figure 1, shows the various elements of the ROCS architecture.

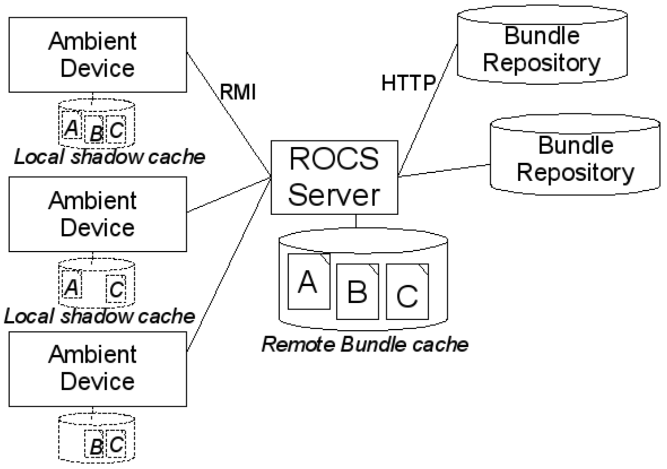


Fig. 1. ROCS Architecture

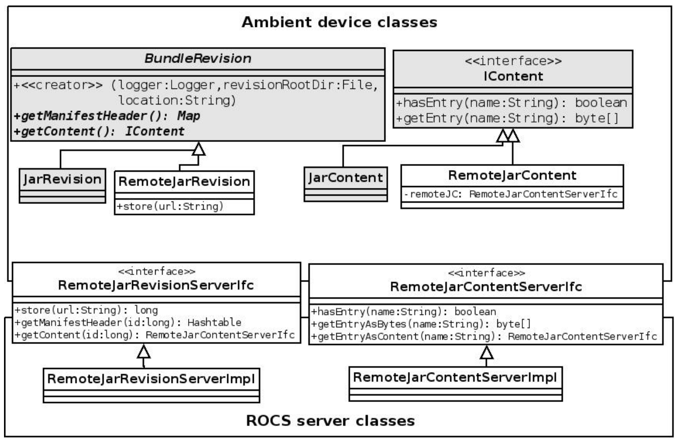
B. ROCS server design

As we want to keep the standard OSGi infrastructure, we introduce an extension to the bundle installation procedure.

[1]In our implementation, we use RMI, but any transport mechanism can be used such as raw TCP, UDP, HTTP, or SOAP. The only requirement is to be able to transport primitive types and arrays of primitive types

When installing a bundle, the ambient device issues a command that indicates the URL from which the bundle can be retrieved. The OSGi framework uses various handlers to manage URL schemes (e.g., `http:` or `file:`). In our case, we added a handler for a new `reference:http:` URL scheme. When the OSGi framework encounters such a scheme it automatically uses the ROCS server. We first describe the ROCS architecture additional classes, then we describe how a bundle is installed and when a required bundle resource is resolved.

ROCS additional classes. As illustrated on Figure 2, the bundle management architecture relies on two interfaces: `BundleRevision` and `IContent`[2].



Ambient device classes
BundleRevision
+<<creator>> (logger:Logger,revisionRootDir:File, location:String)
+getManifestHeader(): Map
+getContent(): IContent
<<interface>>
IContent
+hasEntry(name:String): boolean
+getEntry(name:String): byte[]
JarRevision
RemoteJarRevision
+store(url:String)
JarContent
RemoteJarContent
-remoteJC: RemoteJarContentServerIfc
<<interface>>
RemoteJarRevisionServerIfc
+store(url:String): long
+getManifestHeader(id:long): Hashtable
+getContent(id:long): RemoteJarContentServerIfc
<<interface>>
RemoteJarContentServerIfc
+hasEntry(name:String): boolean
+getEntryAsBytes(name:String): byte[]
+getEntryAsContent(name:String): RemoteJarContentServerIfc
RemoteJarRevisionServerImpl
RemoteJarContentServerImpl
ROCS server classes

Fig. 2. The ROCS class diagram. Classes in dark exist in Felix; classes in white are our contributions.

The `BundleRevision` interface represents the access to the bundle meta-data and its content. The meta-data information enables the creation of the `ModuleLoader` class loader that isolates the bundle classes, and gives a reference to an `IContent` object. The `IContent` interface represents accesses to the bundle content. In order to build a remote access to these interfaces, we build peer interfaces on the ROCS servers. `BundleRevision` is paired with `RemoteJarRevisionServerIfc` and `IContent` is paired with `RemoteJarContentServerIfc`. The `RemoteJarRevisionServerIfc` has one additional method, `store(URL string);` that asks the ROCS server to download the Bundle from the repositories. Once the method has been applied, the framework works transparently since all other methods are peered with remote ones. On the ambient framework side, we override the standard implementation classes `JarRevision` and `JarContent` with `RemoteJarRevision` and `RemoteJarContent`, which are proxies of the remote ROCS server. On the server side `RemoteJarRevisionImpl` and `RemoteJarContentServerImpl` are the servers that manage access to bundle classes and resource byte arrays.

[2]These interfaces names come from the Apache Felix OSGi implementation. We also implemented the same architecture on the alternative Concierge OSGi implementation.

Bundle Installation. When installing a new bundle, the framework identifies the `reference:http` scheme and triggers the sequence illustrated in Figure 3.

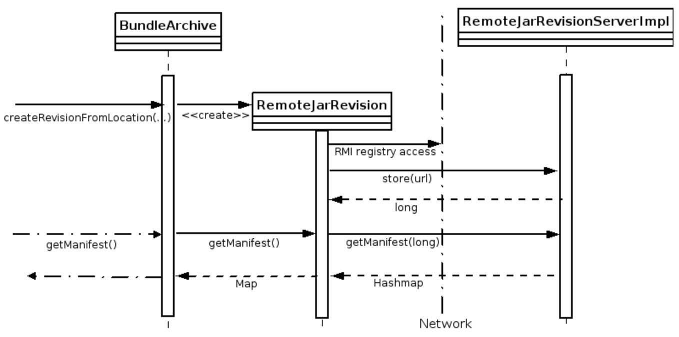


Fig. 3. Sequence diagram: remote bundle installation.

The `RemoteJarRevisionServerImpl` is always alive and the framework `BundleArchive` class creates the `RemoteJarRevision` instance which is the stub to the `RemoteJarRevisionServerImpl`. The first method to be called is `"store(URL string)"`, and returns a `long` value representing the bundle ID. This id is inserted as a parameter for the remaining accesses to the remote bundle. During this step, the framework creates the shadow bundle cache on the ambient device and controls that the bundle satisfies the required conditions to be installed. Once installed, remote bundle resources can be resolved.

Bundle Resources Resolution. When the framework needs to access resources from the bundle it triggers the sequence diagram shown in Figure 4.

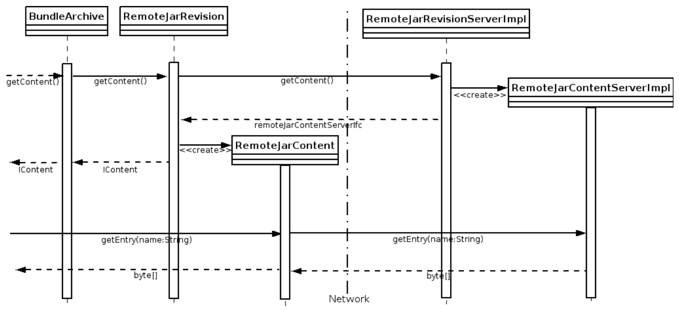


Fig. 4. Sequence diagram: remote bundle resolution.

The sequence creates a `RemoteJarContent` instance, which is a proxy of the `RemoteJarContentServerIfc` service. Each time a new class is needed, the standard class loading mechanism is launched. If the class comes from the current `ModuleLoader` installer, it makes a call to the `"getEntry(String name)"` remote method of `RemoteJarContentServerIfc`. The byte array that is returned is defined as a Java class using the framework standard mechanism. The entire remote mechanism is transparent to both the OSGi and the transport mechanism that we use between the ambient device and the ROCS server. With this approach, all resources are directly loaded into the virtual machine memory without having to be first stored on the local device storage.

C. Designing a minimal runtime environment for booting OSGi framework

Standard Java classes consume a sensible amount of disk space: around 35 MB for IBM Blackdown 1.4, 40 MB for Sun JVM 1.5, and 9 MB for GNU Classpath 0.93. These classes are too big to fit on an ambient device. Our solution is to use the ROCS framework to handle standard classes that are not necessary at boot-time. We boot Java/OSGi with a minimalist local classpath, and then install a *classpath bundle* containing all of the standard Java classes which it then re-exports. Using this approach, standard classes are also isolated in a bundle class loader: each time another bundle needs to access a standard class, it needs to explicitly import the corresponding Java package, which is from our point of view an advantage since we know exactly which packages are being used.

This approach raises two questions. How can we extract the minimal boot classpath? Is it working for `java.*` packages?

Extracting the minimal boot classpath. As a first empirical approach, we designed a script that generates the classpath bundle automatically. The script starts the OSGi framework with a remote classpath bundle that exports no package. The script logs all classes which are loaded during boot time with the virtual machine `verbose:class` option. From that list, it computes the needed packages list, builds the Jar file for the local classpath, and builds the OSGi manifest file with all the required classes.

Relaxing the java.* limitation. This limitation is only expressed in the Java Security Architecture specifications: "Due to historical reasons, all [`java.*`] classes have a class loader that is [the primordial class loader]."[3]. Both the language and the JVM specifications do not specify any instructions regarding this topic. Our experiments show that this behavior is not enforced by the virtual machines we tested, but only by OSGi frameworks. We patched this limitation in our OSGi implementation and enabled classes from `java.*` packages to be provided by standard OSGi bundles. We also patched the OSGi implementation in order to resolve standard classes from the classpath bundle before searching in the local classpath. This approach suggests that we seek new classes from the network before their local implementation, which enables to bypass the resource limitations in the ambient environment dynamic use-case.

Our architecture is able to bootstrap a minimal OSGi/Java runtime environment in an ambient device and run any bundle / any classes downloaded from remote ROCS servers. In the next section we present our evaluation of the ROCS architecture.

[3]Section 5.2 from `http://tinyurl.com/javasecurity`

IV. Evaluation

We carried out many evaluations of the ROCS architecture to show that it is both a portable and an efficient solution. We first present the test setup we uses, then we present hard drive storage improvements and finally we present a response time analysis of the architecture.

A. Test configuration

We tested the ROCS framework on two different hardware environments: the LinkSys NSLU2[4] and the iPaq 5550[5]. Both devices share the same kind of specification:

- ARM CPU running at 266MHz on the NSLU2, 400MHz for the iPaq,
- 32MB of SDRAM for memory for the NSLU2, 128MB for the iPaq,
- 8MB of Flash memory for the NSLU2 local storage, 48MB for the iPaq, and
- Ethernet / Wifi controllers.

We made all our tests on the NSLU2 equipment since it is the most constrained device, but we checked portability of our approach as we made the same tests on the iPaq equipment. It is also worth noting that these kinds of ambient devices have more RAM memory than storage space. As far as the software unit is concerned, we used the following stack:

- Linux 2.6.21 kernel and GNU libc from OpenEmbedded project[6],
- JamVM 1.4.5[7] and both GNU-classpath 0.93[8] and Sun-JRE1.5 classpath,
- A patched version of Apache Felix[9] and Concierge[10] available on demand
- A testing profile containing 13 bundles: 4 "standard" bundles, 5 bundles to enable remote management (`mosgi` package), and 4 bundles for MBean management probes (`mosgi.managedelements` package). This testing profile is detailed in table I, which gives each bundle storage size in KB. Altogether, bundles represent a storage space of 473KB.

Using two types of OSGi implementations, two hardware equipment units and two sets of standard classes, we show that the ROCS architecture is portable and independent from the software stack. In the following section, we are only considering results using Apache Felix, GNU Classpath and the LinkSys equipment. We will now show that the 8MB local storage capacity is sufficient to run the environment.

B. Hard drive consumption

The storage consumption must be shared between a fixed size that corresponds to the minimal bootstrap system size and a growing storage size that corresponds to new bundles

[4]http://en.wikipedia.org/wiki/NSLU2
[5]http://tinyurl.com/ipaq5550
[6]http://www.nslu2-linux.org/wiki/Development/MasterMakefile
[7]http://jamvm.sourceforge.net
[8]http://www.gnu.org/software/classpath/
[9]http://felix.apache.org
[10]http://concierge.sourceforge.net/

Bundle	Size (kB)
...shell.jar	50
...shell.tui.jar	11
...bundlerepository.jar	129
...log.jar	22
...mosgi.jmx.remotelogger.jar	12
...mosgi.jmx.agent.jar	99
...mosgi.jmx.registry.jar	13
...mosgi.jmx.rmiconnector.jar	83
...mosgi.console.ifc.jar	9
...mosgi.managedelements.osgiprobes.jar	11
...mosgi.managedelements.attributeaccessor.jar	5
...mosgi.managedelements.attributetester.jar	7
...mosgi.managedelements.attributeviewer.jar	15
Total	**473**

TABLE I
TEST PROFILE (SET OF BUNDLES)

deployment. The experiment is conducted here on the LinkSys NSLU2 device. Table II shows the considered initial environment.

Layer	Size (kB)
...OS	5 000
...JamVM runtime	500
...Apache Felix classes	350
...Standard Classpath classes	9 000
...13 Bundles	500
Total	**15 350**

TABLE II
STORAGE PLACE USED BY A STANDARD OSGI/JAVA STACK

When using the ROCS framework, the standard classpath is reduced to the minimal set of classes needed to boot the entire OSGi/Java stack (2000 bytes), and the various bundles are shadowed on the ambient device. A shadow bundle costs about 400 bytes. Table III shows the storage we obtained with ROCS.

Layer	Size (kB)
...OS	5 000
...JamVM runtime	500
...Apache Felix classes	350
...Standard Classpath classes	2 000
...13 Bundles	5
Total	**7 855**

TABLE III
STORAGE PLACE USED BY THE OSGI/JAVA STACK USING ROCS

The fixed size is reduced from 14.850MB to 7.850MB which is below the 8MB limitation, and 150kB is left to install new bundles. The size of cached bundles, which is expected to grow throughout the framework life-cycle, has an initial size of 400 bytes corresponding to the Classpath Bundle shadow cache. Since each shadow bundle costs 400 bytes we can theoretically install more than 300 bundles (we have not tested this limitation). The overall fixed gain is 48% in disk space consumption, but if we only consider GNU Classpath, we can ignore 67% of the classes when booting the framework (88% with sun JRE1.5). It is also important to note that the 5MB corresponding to the operating system are not optimized, as we left many Linux management applications running for debugging purposes, and we can still gain some storage place. However, this out of the scope of this paper.

At this stage, the LinkSys NSLU2 is able to boot and run a fully compatible Java/OSGi stack as well as many bundles managed by the ROCS server. The bootstrap and remote classes access delays are still an issue that we investigate and discuss hereafter.

C. Performance

Since we cache bundles on a remote server, we may presume longer startup times, and a degraded overall deployment performance. In our tests described below, results are much different than initially expected. They emphasize the relevance of ROCS as a basis for building ambient environments.

We evaluated 3 durations in the OSGi/Java life-cycle. The first one is the framework initial startup. The second corresponds to bundle installation. The third one corresponds to class loading. Finally we run the benchmark to compare the architecture with and without ROCS. When using the ROCS framework, the server is hosted on a PC connected to a common 100 MB/s Fast local Ethernet network.

Starting the framework. This step occurs between the entry of the OSGi framework's `main` method and the first application bundle is installed. Using ROCS does not impact starting time with the Felix implementation.

Installing bundles. This step includes loading jar files on ROCS servers, from local URL, and installing shadow bundles on the ambient device. With JAR files on the local disk, between 40 and 600 milliseconds are used, for bundles ranging between 10 KB and 130 KB. With jar files on the ROCS server, 100 ms are needed, only once, to obtain a reference on the remote RMI server, but only once. Then, between 15 ms and 20 ms per bundle are used, for bundles with the same sizes as above. Actual measurements of these per-bundle delays are plotted on Figure 5; we used sample bundles from the Apache Felix source repository, with incremental sizes coming from our testing profile. Results are similar with both the LinkSys NLSU2 hardware and the iPaq 5550.

Since only the meta-data are transferred over the network, the installation of a shadow bundle with ROCS has a constant duration (lower curve). When installing the entire bundles, the duration is proportional to the bundle size. We also compare the installation time of the bundles both on the local flash disk and on an external USB storage key, and we see that the native flash driver is very slow.

Loading classes. This step includes, for each bundle in the test profile, loading all classes that are needed directly or indirectly.

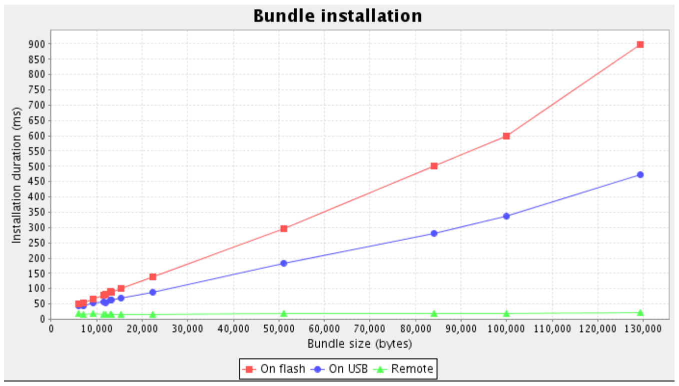


Fig. 5. Time needed to install a bundle depending on its size.

Figure 6 shows the time needed to load a class, depending on its size. Measures were obtained using specially crafted bundles, each containing 500 classes of a fixed size (on the x axis). Classes were loaded using `Class.forName()`, so that we know when they are loaded without interferences from potential optimizations by the Java Virtual Machine. Results on Figure 6 are averages and mainly represent the resource transport costs between ROCS servers and ambient devices.

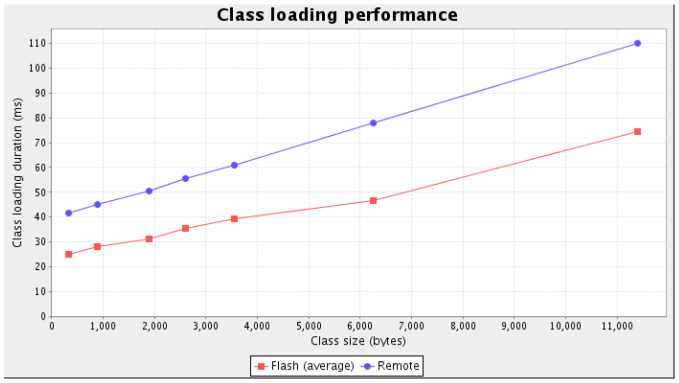


Fig. 6. Class loading time: local vs. remote

In the standard Java classpath, around 75% of classes are below 4 kB, which gives us a sensible estimation of common class sizes. Using this estimation, Figure 6 shows that loading a class is 20 milliseconds slower when we use RMI, compared to loading it locally.

A simple benchmark. We evaluated the startup time for the framework along with the 13 bundles from our test profile. Measures are performed on Apache Felix with and without using the ROCS cache server. Table IV shows how much time is spent between starting up the framework, installing bundles and resolving their dependencies. The last columns shows the overall gain.

ROCS	off (s)	on (s)	Gain (%)
Startup	1.9	1.9	0
Install	12	6.2	48
Resolve	15	15.3	-2
Total time	**29**	**23.4**	**19**

TABLE IV
ROCS BENCHMARKS

It is worth noting that our results are better than the ones obtained on the same evaluation we made on the Concierge [13] implementation. We see that the real gain is on the installation time and that the transport cost is rather negligible. OSGi on a LinkSys NSLU2 device benefits from using the ROCS server. Of course, this stands provided the following requirements are met: (1) local storage has a slow write throughput (e.g. NOR Flash memory [14]), and (2) network access is Ethernet grade, and (3) the remote server has storage with casual PC grade access delays.

Last but not least, once classes have been loaded, the ambient device virtual machine classes behave "normally", meaning that there is no further performance penalty during classes execution.

V. Conclusion

This article presented ROCS, a framework for running OSGi/Java stacks on ambient devices without locally installing the real bundles class definitions, but only shadow representatives. The architecture relies on a proximity application server that delivers the required resources. We showed that when running the framework in a dedicated network, benefits are obvious and standard JavaSE applications can be run.

ROCS main use-case is to enable Java/OSGi environments for modems and set-top boxes, in the context of smart homes and multiple home service providers. In such setup, the ROCS server is hosted by the connectivity provider, on the DSLAM. This equipment unit already connects and manages modems from a small area to the operator network [15]; with ROCS it also provides them software facilities. ROCS architecture enables the deployment of constraint devices with a read-only boot partition and a rather small read-write partition for the cache part. These equipment units can store their configuration somewhere on the Web and get their new applications in a highly dynamic manner.

ROCS is energy-aware since only the required resources are installed. If the costumer uses every function of a specific application, we have the same cost as a locally installed application would have. If a user only launches a tiny subset of the functions, their corresponding resources won't be downloaded: if the user installs a PDF document viewer but does not need printing, the corresponding functions subset is never installed neither locally nor in memory.

ROCS architecture stresses the point of designing new ambient operating systems as cooperations between constraint small equipment units and remote servers.

ROCS architecture provides many promising perspectives since we can make many improvements in the general architecture. Among these improvements, we can share bundle class delivery between many ambient devices, we can provide a good level of privacy since shadow cache can still contain private data, we can improve and make agnostic the transport protocol using raw TCP or any high-level transportation, and so on.

One benefit on precisely managing resource downloading is to enable real usage observation and real cost accounting since only strictly required functions are eventually downloaded. What we have yet to explore are smart class loaders that can cope with networking faults. We are aware of these issues and we are working on *Delayed and Disrupted Tolerant Network* and P2P layers for efficiently accessing resources.

Finally, it is important to note that our approach could be trivially extended to other frameworks, provided that they are either based on component approaches, and that they use class loading mechanisms. For instance, it can be developed for lightweight, shared J2EE servers infrastructure.

REFERENCES

[1] I. Sun Microsystems, "Introduction to cloud computing architecture," White Paper, Sun Microsystems, Inc., Tech. Rep., Jun. 2009.

[2] OSGi Alliance, *OSGi Service Platform, Core Specification, Release 4, Version 4.1*, 4th ed. OSGi Alliance, 2007. [Online]. Available: http://www.osgi.org/

[3] C. Bethell, "Open source os: The future for mobile?" Juniper Research, Tech. Rep., Jul. 2009.

[4] I. Sun Microsystems, "J2me building blocks for mobile devices," White Paper, Sun Microsystems, Inc., Tech. Rep., May 2000.

[5] J.-Y. Tigli, S. Lavirotte, G. Rey, V. Hourdin, and M. Riveill, "Lightweight Service Oriented Architecture for Pervasive Computing," *International Journal of Computer Science Issues (IJCSI)*, vol. 4, no. 1, 2009.

[6] A. Courbot, G. Grimaud, J.-J. Vandewalle, and D. Simplot-Ryl, "Application-driven customization of an embedded java virtual machine," in *Second International Symposium on Ubiquitous Intelligence and Smart Worlds (UISW2005)*, S.-V. LNCS, Ed., Nagasaki, Japan, December 2005.

[7] G. Thomas, F. Ogel, A. Galland, B. Folliot, and I. Piumarta, "Building a flexible Java runtime upon a flexible compiler," *International Journal of Computers & Applications*, vol. 27, no. 1, pp. 27–34, 2005.

[8] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java on the bare metal of wireless sensor devices: the squawk java virtual machine," in *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*. New York, NY, USA: ACM, 2006, pp. 78–88.

[9] D. Rayside, E. Mamas, and E. Hons, "Compact java binaries for embedded systems," in *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 9.

[10] Apache Harmony, "Open Source Java SE," http://harmony.apache.org/, 2003–2008.

[11] S. Liang and G. Bracha, "Dynamic class loading in the Java virtual machine," in *ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, Canada, 1998, pp. 36–44. [Online]. Available: http://cs-www.cs.yale.edu/homes/liang-sheng/

[12] C. Franke and P. Robinson, "Autonomic provisioning of hosted applications with level of isolation terms," *ease*, vol. 0, pp. 131–142, 2008.

[13] J. S. Rellermeyer and G. Alonso, "Concierge: a service platform for resource-constrained devices," in *EuroSys '07: Proceedings of the 2007 conference on EuroSys*. New York, NY, USA: ACM Press, 2007, pp. 245–258.

[14] T. Corp., "NAND vs. NOR Flash memory - technology overview," April 2006.

[15] Y. Royon and S. Frénot, "Multiservice home gateways: Business model, execution environment, management infrastructure," *IEEE Communications Magazine*, vol. 45, no. 10, pp. 122–128, October 2007.