



DB3.6 – Virtual Machines for Embedded Environments

Stéphane Frénot
stephane.frenot@inria.fr

Serafeim Papastefanos
serafeim@telecom.ntua.gr

Identifier:	Deliverable D B3.6
Class:	Prototype
Version:	2.11
Version Date:	25/10/2007 (public version: 14/01/2008)
Distribution:	Public
Responsible Partner:	INR

Filename: WPB3_00001_v2.11_DB3.6.odt

DOCUMENT INFORMATION

<i>Project ref. No.</i>	IST-6thFP-026442
<i>Project acronym</i>	MUSE
<i>Project full title</i>	Multi-Service Access Everywhere
<i>Security (distribution level)</i>	Public
<i>Contractual delivery date</i>	M 42
<i>Actual delivery date</i>	25/10/2007 (public version: 14/01/2008)
<i>Deliverable number</i>	DB3.6
<i>Deliverable name</i>	Virtual Machines for Embedded Environments
<i>Type</i>	Prototype
<i>Status & version</i>	V2.11
<i>Number of pages</i>	18
<i>WP / TF contributing</i>	WPB3
<i>WP / TF responsible</i>	Sam D'Haeseleer
<i>Main contributors</i>	Stéphane Frénot
<i>Editor(s)</i>	Stéphane Frénot
<i>EU Project Officer</i>	Pertti Jauhiainen
<i>Keywords</i>	Java, OSGi, Embedded device
<i>Abstract (for dissemination)</i>	This deliverable presents MUSE WPB3 activities around the home gateway and more precisely a Home Gateway running a Java virtual machine and an OSGi framework. The stress is on making the Execution Environment fit in a resource-constrained Home Gateway.

DOCUMENT HISTORY

Version	Date	Comments and actions	Status
1.0	12/09/2007	Creation of document	Draft
2.0	1/10/2007	Addition of a new chapter	Draft
2.7	9/10/2007	Added results values	Draft
2.8	22/10/2007	Temporary revision	Deliverable
2.9	24/10/2007	Deliverable	Deliverable
2.10	25/10/2007	Small updates	Deliverable
2.11	14/01/2008	Converted to a Public version, in agreement with all WPB.3 Partners	Deliverable

TABLE OF CONTENTS

DOCUMENT INFORMATION.....	2
DOCUMENT HISTORY.....	3
TABLE OF CONTENTS.....	4
LIST OF FIGURES AND TABLES	5
ABBREVIATIONS.....	6
REFERENCES.....	7
EXECUTIVE SUMMARY.....	8
1 INTRODUCTION.....	9
2 THE RUN-TIME ENVIRONMENT.....	9
2.1 The NLSU2 embedded device.....	9
2.2 Operating systems.....	10
2.3 Java virtual machines.....	10
2.4 OSGi frameworks.....	10
2.5 Conclusion.....	11
3 A NO-CACHE OSGI IMPLEMENTATION – THE ROCS.....	11
3.1 OSGi bundle caching.....	11
3.2 A remote bundle server.....	11
3.2.1 <i>The profile manager server specification</i>	12
3.2.2 <i>The profile server specification</i>	12
3.2.3 <i>A typical session</i>	13
4 PERFORMANCE RESULTS.....	14
5 CONCLUSIONS.....	15
5.1 Why does it work ?.....	15
5.2 Current Limitations.....	16
ANNEXE A : UPNP/TR069 PROXY IN CONSTRAINED ENVIRONMENT.....	17

LIST OF FIGURES AND TABLES

FIGURE 1: ROCS ARCHITECTURE.....	12
FIGURE 2: ROCS PROFILE SERVER INTERFACE	12
FIGURE 3: ROCS JAR SERVER INTERFACE	13
FIGURE 4: GETRESSOURCEURL SEQUENCE DIAGRAM.....	13
FIGURE 5: METHOD CALL SEQUENCE DIAGRAM IN ROCS.....	14
FIGURE 6: UPNP/TR69 JAVA STACK	18
TABLE 1: ROCS COMPARATIVE RESULTS.....	15
TABLE 2: INSTALL AND STARTING STAGES COMPARISON.....	15

ABBREVIATIONS

JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
MOSGi	Managed OSGi
RMI	Remote Method Invocation
ROCS	Remote OSGi Cache Service
VOSGi	Virtual OSGi
VM	Virtual Machine

REFERENCES

- [Concierge] Concierge: an optimized OSGi framework, <http://concierge.sourceforge.net/>
- [DB3.4] MUSE deliverable : "Specification of Residential Gateway configuration"
- [Felix] OSGi R4 implementation from apache, <http://felix.apache.org/>
- [GentooSlug] How to Install Gentoo on NSLU2, http://gentoo-wiki.com/Gentoo_on_NSLU2
- [GnuCP] The GNU Classpath implementation, <http://www.gnu.org/software/classpath/>
- [IEEE] Yvan Royon, Stéphane Frénot, "Multiservice Home Gateways: Business Model, Execution Environment, Management Infrastructure", IEEE Communications Magazine, October 2007
- [IEEE2] A. Nikolaidis, S. Papastefanos, G. Doumenis, G. Stassinopoulos, M.P. Drakos, "Local and Remote Management Integration for Flexible Service Provisioning to the Home", IEEE Communications Magazine, October 2007
- [JamVM] A compact Java Virtual Machine <http://jamvm.sourceforge.net/>
- [JRE] Sun Java Run-Time Environment, <http://java.sun.com/>
- [Knopflerfish] Knopflerfish OSGi implementation, <http://www.knopflerfish.org/>
- [MOSGi] Managed OSGi, <http://felix.apache.org/site/mosgi-managed-osgi-framework.html>
- [NSLU2] NLSU2 reference chart, <http://tinyurl.com/2ot7n9>
- [OpenOSGi] Open Source OSGi implementations, <http://www.osgi.org/markets/opensource.asp>
- [OSGi] OSGi alliance, <http://www.osgi.com>
- [SABLEVM] An open-source virtual Machine, <http://www.sablevm.org/>
- [SLUGOS] An operating system for the slug equipment, <http://www.nslu2-linux.org/wiki/SlugOS/SlugOSBE>
- [UNSLUNG] An operating system for the slug equipment, <http://www.nslu2-linux.org/wiki/Unslung/HomePage>
- [SLUGFIRM] A flashable bios for the SLUG equipment, <http://www.nslu2-linux.org/wiki/FAQ/FirmwareMatrix>
- [VOSGi] Virtual OSGi project, <http://gforge.inria.fr/projects/vosgi>

EXECUTIVE SUMMARY

This deliverable presents MUSE activities around embedded environments for home gateways and more precisely a Java virtual machine and an OSGi framework stack.

In this deliverable we study the possibility of implementing a full OSGi/Java stack within an embedded device that does not have sufficient disk space. Java standard classes are growing each new release and even old Java implementations (1.3) are too big to fit in embedded environments. We propose the ROCS architecture (Remote OSGi Cache Storage) that caches bundles on a remote server and provides classes when necessary. With ROCS the initial file provisioning on the gateway is minimal and can be stored on the classical 8MB limits. The remaining classes only exist in main memory whose 32MB limits are more tolerant.

The deliverable presents the run-time environment initial provisioning, and the ROCS framework. Performance results from the proof-of-concept indicate that with the ROCS solution (1) many services can be “consumed” without the need for more expensive storage, and (2) with the current bandwidth available in residential networks it even outperforms a classical solution with more expensive storage.

In addition to the previous, we include an annex concerning the execution of the UPnP/TR069 Proxy Java implementation in a constrained environment. It describes the steps that were taken in order to run the UPnP / TR069 Proxy in the embedded device that is used throughout the deliverable. It stresses the difficulties we had to find a suitable and embeddable full Java stack.

1 INTRODUCTION

Current set-top box environments are running native operating systems (Linux or proprietary ones) but current trends lead to a necessity to integrate new services. On native environments services can be developed in plain old C nevertheless the Java language shortens development time and simplifies designed architectures. The OSGi framework enables new possibilities in Java software development. The first one is the service oriented programming paradigm, that enables a rapid development of applications through composition and reuse, and the second one is a multi-application view on top of the Java Virtual Machine.

Unfortunately current full OSGi/Java/Linux stacks do not fit into embedded device flash disks. The typical MUSE Home Gateway owns an 8MB disk, while Java ClassPath weighs a minimum of 9MB.

This document shows the ROCS project details which allows to serve a full OSGi/Java stack for this environment. ROCS relies on Java intrinsics abilities to download classes and make transparent remote procedures calls through RMI.

Next section details the run-time environment we choose for our tests, then we present the ROCS architecture. We test this architecture in a testbed environment and we present in section 4 the corresponding results. We finally propose some comments in the conclusion.

2 THE RUN-TIME ENVIRONMENT

MUSE deliverables DB3.4, DB3.5 and the demonstrator in MB4.3 describe a managed run-time system for a multi-service multi-provider environment. Management issues of this infrastructure are described in [IEEE].

The service gateway run-time hosts an OSGi service platform and some standard services. The services we implemented on top of the home gateway are MOSGi and VOSGi. MOSGi is a management layer that enables remote and local management of Home Gateways. VOSGi (for virtual OSGi) is a layer that enables to run multiple instances of OSGi within a core instance. All these services need to run in the main memory of the residential gateway equipment and are initially installed on the equipment storage disk.

2.1 The NLSU2 embedded device

We use the LinkSys'[NLSU2] equipment which main characteristics are :

Intel IXP420 (ARMv5TE) CPU running at 266MHz

- 32MB SDRAM
- 8MB NOR flash
- 10/100 ethernet (builtin)
- 2x USB 2.0 host

This equipment has the same processing power and available memory as the MUSE residential gateway specification. It has another very interesting feature since there are many projects and web pages about it: it is very easy to find information.

2.2 Operating systems

Linksys company has opened the BIOS code of its equipment. It is then possible to install a specific BIOS and a Linux operating system. This manipulation is clearly presented in [SLUGOS]. We are currently working on a Gentoo based system, with a 2.6.14.3 kernel. [GentooSlug]. The kernel has a size of 1.6MB and the corresponding file system has a size of 3.5MB. The kernel is not optimized for embedded environments and the file system hosts applications that should not be included in a “real” environment (remote connection, profiling tools...) but it gives a good idea of an initial size. A current state of the art of slug running operating systems is available on [SLUGFIRM].

On top of the operating system we need to run a Java virtual machine.

2.3 Java virtual machines

Many Java virtual machines are said to run on embedded environments but few of them actually do with all necessary functionalities. Standard virtual machines from Sun Microsystems do not run on ARM powered equipments. Having a clear vision of which virtual machines are running on embedded device is rather impossible because most of them are only tested in dedicated environments and do not work if the environment slightly differs. We tried the following virtual machines:

- IBM's J9,
- Kaffe,
- Wonka,
- JamVM+GnuClassPath

Many virtual machines for embedded environments are discontinued projects (wonka, kaffe). IBM J9 has the drawback of being a closed source project. We finally focus on the JamVM [JamVm] / GnuClassPath [GnuCP] solution since it is a recent implementation which only targets the byte-code interpreter without the burden of implementing standard classes. The standard classes are provided by the gnu-classpath project.

The JamVM virtual machine takes another 300kB and the initial gnu-classpath has 9MB available size. In order to reduce the size of the initial classpath, we identified the required classes to boot the OSGi framework and we extracted an initial subset of 1.3MB.

On top of the JamVM virtual machine we are running an OSGi R3 framework.

2.4 OSGi frameworks

Current open-source OSGi implementations [openOSGi] are not targeted at embedded environments. That means that even though they can run on a jdk1.3 compatible environment (JamVM/GnuClasspath for instance) they do not try to optimize their run-time consumption. [Concierge] is an OSGi open-source implementation project that tries to run in a very small foot-print (90 kB). We used this implementation to run our prototype.

2.5 Conclusion

We described the run-time environment as a layered approach using the NSLU2 hardware equipment, the Gentoo operating system, the JamVM/GnuClasspath virtual machine and the Concierge OSGi implementation. We run our MUSE MOSGi/VOSGi implementation on this architecture. At this point 6.3MB out of 8 are already used on the disk. Each time a new OSGi bundle is installed, it is initially locally copied and run from the local environment. The 1.7MB remaining on the disk is not sufficient to hold new bundles. In the next section we present an extension to the Concierge framework in order to have a zero length local cache of bundles. This extension should also enable to download additional classes that were not in the initial subset of 1.3MB of classes.

3 A NO-CACHE OSGI IMPLEMENTATION – THE ROCS

The Remote OSGi Cache Storage (ROCS) is a RMI based service for OSGi frameworks that avoids to locally cache running bundles. A shadow cache is present on the gateway but it contains neither classes nor resources: only the profile structure directory is present. The necessary resources and classes are brought back to memory from the ROCS server through RMI calls. We maintain the profile structure for two reasons. The first reason is to keep the initial framework design the most compatible with our modifications – It should compile without the burden of patching the entire framework, so we only extract a subset of the profile management layer. The second reason is to improve only the strict necessary elements – Empty local profiles do not cost much place, but having a systematic remote call should have cost too much network bandwidth.

3.1 OSGi bundle caching

We suppose that the reader knows the main concepts of OSGi. A manager installs and starts new bundles. The bundles are Java archives packaged in a specific manner. They are downloaded from a remote location and locally installed in a repository. This local repository is stored on the local file system and is identified by a profile ID (called the user profile). In most implementation the local profile is `.<osgiImplem>/<profileName>/`. Each profile contains all corresponding installed bundles. The various frameworks associate a specific classloader to each bundle, which has the responsibility to load the requested classes on-demand, resources and native code from the archives stored in the local repository.

An embedded device has a very small local disk (8 MB), so that approach do not satisfies us. We thus propose a solution where we remotely cache bundles.

3.2 A remote bundle server

One of the most powerful features of the Java virtual machine is the ability to populate the environment with new remote classes. This ability is showed in applets and RMI code-stub downloading Java features. Nevertheless in OSGi this feature is not sufficiently exploited. We extended the bundle classloader to interact with a remote server that serves the bundle classes. Figure 1 illustrates our architecture. When a new bundle has to be installed, the home gateway contacts a remote ROCSserver and asks for its installation. Each time the running application needs to access a class from the remote server, a request is sent to its corresponding ROCS server. The byte array corresponding to the class is transferred to the class loader, which then can define the new class in the virtual machine.

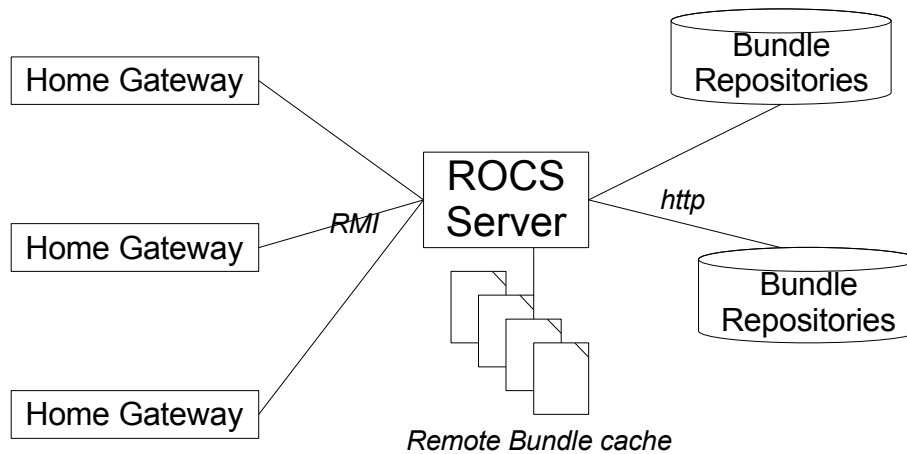


Figure 1: ROCS Architecture

The ROCS server relies on two interfaces. The first one is a profile manager server, the second one is a bundle cache server.

3.2.1 The profile manager server specification

When the profile is determined on the gateway, it asks the profile manager server a reference on its specific bundle class server. Even though most current OSGi implementations are running with a mono-profile paradigm in mind, our VOSGi proposal (virtual OSGi) has to enable the coexistence of many OSGi profiles (one for each service provider). The server API is the following.

```
public interface ROCSProfileServerIfc extends Remote {
    public ROCSJarServerIfc getProfile(InetAddr myIp,
        String profileName) throws RemoteException;
    public void removeProfile(String fileName)
        throws RemoteException, ProfileNotFoundException;
}
```

Figure 2: ROCS Profile Server Interface

The API enables the creation, and destruction of profiles and is mainly used to get a reference to a running profile server.

3.2.2 The profile server specification

Once a gateway profile is “connected” to its remote server it will make a deployment structure corresponding to the installed bundle. Only the directory structure is created but the bundle jars (main bundle jar and embedded jars (see OSGi specifications)) remain on the remote site.

The corresponding server API is the following:

```

public interface ROCSJarServerIfc extends Remote {
    public Hashtable getManifest(String bundleName) throws
RemoteException;
    public byte[] getBytes ( String jarPath , String classpath ,
String filename) throws RemoteException;
    public void store (String file, byte[] localBundle) throws
RemoteException;
    public void store (String file, URL url) throws RemoteException;
    public URL getResourceURL (String jarPath, String classpath,
String filename) throws RemoteException;
}

```

Figure 3: ROCS Jar Server Interface

When a new bundle is installed, the gateway asks the remote server to download and store the bundle on the remote site. This is made with one of the two store methods: the bundle is retrieved either locally (filesystem) or from a URL. Then it gets the manifest file to execute the bundle startup procedure: dependencies management, embedded jars extraction, BundleActivator Class instantiation. This is achieved with the getManifest method. The getResourceURL method enables the gateway to get a resource (sound file, image...) from the bundle. Since RMI cannot handle streams, the method redirects the gateway to a secondary web server where it will be able to download or stream the corresponding resource.

The corresponding sequence diagram is show in the next figure.

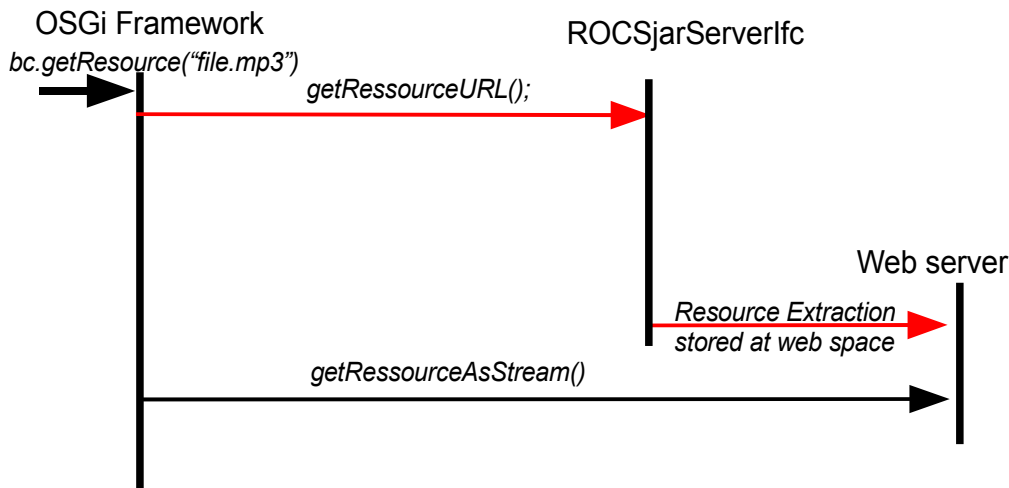


Figure 4: getRessourceURL sequence diagram

The last getBytes method is one of the most important since it is the method that downloads the class bytes and recreates the class definition within the local virtual machine in memory (no disk usage).

3.2.3 A typical session

Figure 2 shows a sequence diagram of calls when a Helloworld bundle is used.

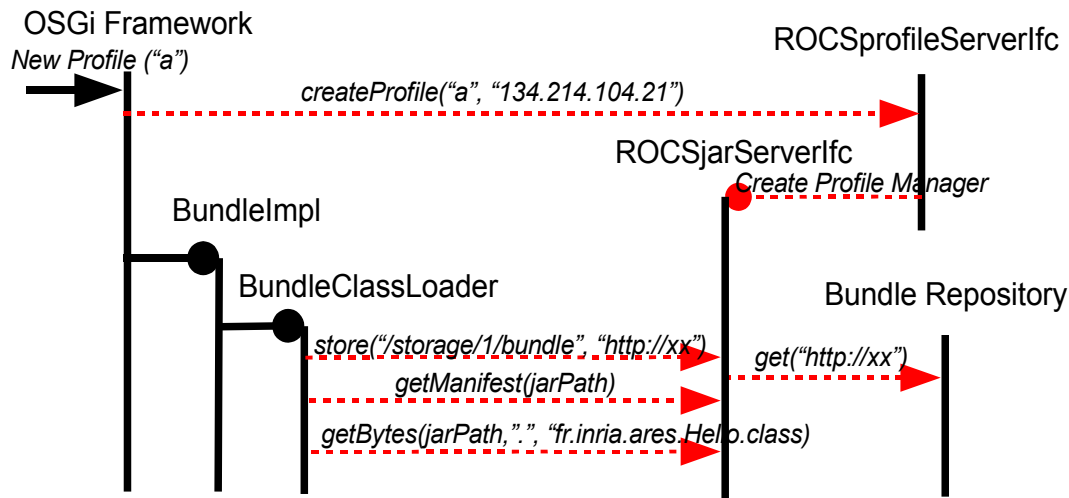


Figure 5: Method Call sequence diagram in ROCS

4 PERFORMANCE RESULTS

We tested ROCS on various conditions. We present here our figures.

- **Initial boot process:** the initial booting process starts when a user powers on the slug and stops before any java application is launched. It correspond to the firmware and operating-system boot.
 - Booting time: 34s
 - Available SDRAM: 20MB/32MB
 - Available Flash: 1.2MB/8MB
 - 1.9 bin
 - 1.8 concierge (+gnu-classpath min)
 - 1.4 usr
 - 0.9 lib (jamvm+others)
 - 0.7 kernel+modules
- **ROCS boot:** rocs booting process corresponds to the jamvm virtual machine, the concierge OSGi framework and management services.
 - Booting time: 8.3s (downloaded 270kB of in-memory classes)
 - Available SDRAM: 14MB/32MB
 - Available Flash: 1.2MB/8MB

We made some comparative tests between a ROCS environment and a environment without it. We designed three profiles. The first one, 'mini' corresponds to the sole system bundle. The second one, 'mana' is the 'mini' profile with some management bundles to load the system (8 bundles that correspond to a disk cache size of 270kB). It corresponds to an empty MOSGi architecture. The third one corresponds to a multi-services gateway and has a cache size of 430kB. The next figure presents the comparative results.

	Mini (0 bundle)		Mana (8 bundles)		Maxi (14 bundles)	
Rocs (on/off)	Off	On	Off	On	Off	On
Local cache size (kB)	~0	~0	290	20	460	30
Free memory (MB)	18.6	16.2	13.3	14	12	13.2
Rocs Boot (s)	0.13	2.4	9.4	8.3	14	11
==> Rocs gain Flash (kB)	0		270		430	
==> Rocs gain Mem (MB)	-2.4		+0.7		+1.2	
==> Rocs gain time (s)	-2.27		+0.9		+3	

Table 1: Rocs Comparative results

The performance gain is always present if we use the ROCS architecture. As soon as we have more than 200kB of cached bundles (which corresponds to 6 bundles in our tests) we have improvement in the three areas: (1) the cache size, (2) the available SDRAM and (3) the boot time. The cache size improvement is related to the main architecture design goal. The available SDRAM improvement is due to the fact that the local deployment of bundle is not any more at the local device duty, and the tested implementation seems to consume more memory than needed. The boot time improvement is linked to the fact that the network bandwidth has better performances than the Flash access and that some activities are now efficiently handled by the ROCS servers.

We made another test series, where we separate the bundle installation process from the bundle starting process. In the bundle installation stage, we measure the time used to validate the bundle: downloading from remote site and controlling its dependencies. In the bundle start stage, we consider the remote downloading of classes (Activator and other). Next figure corresponds to values we observed in the 'Maxi' profile.

	Rocs Off	Rocs On
Framework initialization (s)	0.1	2.4
Bundle Install stage (s)	7.7	2.6
Bundle Starting stage (s)	6.2	6.0
Total	14	11

Table 2: Install and starting stages comparison

Rocs is very valuable during the bundle installation stage (2), since the bundles are downloaded from a remote site and locally controlled and stored. We are twice faster on the remote server than on the home gateway since the remote server makes all the decompression / transmission work on the bundles.

5 CONCLUSIONS

5.1 Why does it work ?

Our architecture has comparable performances as a local cache approach. The main reason is related to the speed of the SDRAM. In the standard approach the bundles are initially downloaded and stored on the local storage disk. Our classes and resources streaming approach avoids this overhead.

Another important element is the inversion of RAM/Flash sizes between embedded and desktop equipment. The available main memory and its low cost drives new equipments with lots of RAM but small disks. The streaming fits with this.

Provided the available network bandwidth is sufficient (~10MB), our approach is faster. We tested ROCS with various numbers of bundles to be installed. The RMI layer has an initial cost of 2.4 s on the slug which is absorbed when we start more than 6 bundles (~260kB of cache). With 14 testing bundles (~430kB) we are 3 s faster with the remote cache than with a local cache.

With the MUSE context the RMI approach suits well since RMI only requires a good connectivity that can be provided by the local DSLAM. In this context the last mile equipment can host the ROCS server and provide a cache for the local bundles.

Finally RMI is not a prerequisite, it is present here only for simplifications. The ROCS system can run on any kind of remote system (SOAP, RPC...) provided it can transport arrays of bytes.

5.2 Current Limitations

Our architecture has some limitations mainly with the classes we removed from the initial bootstrap. Since the 9MB taken by the gnu classpath classes cannot be stored on the home gateway, we had to split the archive into a local part (1.3MB) and another remaining part. In order to access the remaining part, we made a specific bundle accessed through the ROCS server which exports all other standard packages.

There is still a limitation due to the security mechanism of the Java classloader. "Classes from the same package" must be loaded by the same classloader. That means that if one loads `java.util.Vector` from the initial archive all classes from `java.util` must be provided by the same archive. For the moment we restricted the initial archive only with the strictly necessary classes. This leads to run-time errors if a bundle uses a classes whose package is in the initial archive. To alleviate the problem we have three solutions and we are currently evaluating each of them:

- We can extend the size of the initial archive. When a class is needed, we pull out all the corresponding package. In this case the initial archive has a size of about 3.5MB instead of 1.3MB.
- We can optimize the Concierge initial run-time by providing specific implementation for some classes. For instance if it needs `java.util.Vector`, we provide a `rocs.util.Vector` class which is used by concierge so as to leave the `java.util` package on the remote site.
- Modifying the security behavior of the classloader in order to enable the multi-location of the same package. This approach is possible if all bundles are controlled by a closed delivery mechanism such as the one proposed in MUSE.

ANNEX A : UPnP/TR069 PROXY IN CONSTRAINED ENVIRONMENTS

Muse deliverable [DB3.4] included the specifications of a UPnP/TR069 Proxy. This network node was used as an intermediate between the UPnP and TR069 protocols, in order to enable the ACS to remotely access the UPnP devices. More information about this Proxy can be found in [DB3.4] and [IEEE2].

This annex shows the difficulties encountered to build an embedded Java environment. In the following paragraphs the procedure of running this Java implementation of the Proxy in a constrained environment, the LinkSys [NSLU2] device is presented.

As described before, Linksys has opened the firmware code of this device, and many web sites are offering modified versions of it, or even completely new. For the procedure described here, the Unslung [UNSLUNG] firmware was used. This is a modified version of the original firmware of the device which offers lots of additional features like the capability of booting from a USB flash disk (instead of the device's flash memory), downloadable packages and also supports the original Web interface of the device.

In order to observe the behavior of the UPnP/TR069 Proxy in this device, two different Java Virtual Machines were used: JamVM and SableVM. Both virtual machines are open source and fully support the JVM specifications. As described before, Sun's VM does not support the ARM architecture, so only the VMs presented here and these in section 1.3 could be used in the NSLU2 device.

The UPnP/TR069 Proxy was cross-compiled on a PC running Windows and then the various resulting binary classes were moved to the NSLU2 device in order to be run. Firstly, some tests were made with the JamVM. This VM was able to run some simple programs, but was not able to run the UPnP/TR069 Proxy and crashed with a core dump. This was a problem (bug ?) in the JamVM itself and not in the Proxy, and since after some tests the problem could not be isolated the procedure was moved to the SableVM.

The SableVM showed a much better behavior since it was able to run the Proxy until the initialization of the network interfaces. There, a function that enumerated the interfaces of the computer running the Proxy was used – it was needed in order to find the interfaces on which the UPnP part of the proxy would listen for multicast messages. The first signs of the problem showed that this specific function was not supported in the VM. In order to elaborate this guess, the SableVM was used in a 80x86 Linux running PC, and the problem was exactly the same, so the SableVM did not support enumerating the network interfaces. Specifically, the non-supported code was from the CyberLink bundle (which was used for the implementation of the UPnP part of the Proxy). It was from the `org.cybergarage.network` package, in the `HostInterface` class, in the static function `getHostAddress`. What was missing was the static method `getNetworkInterfaces()` of the `java.net.NetworkInterface` class.

In order to overcome this problem, the UPnP/TR069 Proxy was changed to listen to a specific network interface, so no enumeration of network interfaces would be needed. This did not solve the problem though; it just moved it to a different part, since now another networking function (the one needed for the multicast group joining) has been added. The non-supported code was this time also from CyberLink, from the org.cybergarage.upnp.ssd package, HTTPMUSocket class, open function. What was missing was the joinGroup method function of the MulticastSocket class. This function could not be skipped, since without it the UPnP part of the proxy wouldn't be working. In order to implement this function, the SableVM source code needs to be changed in order to include the missing functions needed for UPnP.

After that, the UPnP part of the proxy was disabled, and then the NSLU2 was able to run the Proxy without problems, behaving like a very simple TR069 client and communicating with the ACS.

The problems that were discussed above, which mainly were the result of the non-support of Sun's VM for the ARM architecture and the usage of the OpenSource VMs, did not enable the UPnP / TR069 Proxy to be run on the NSLU2 device. It is expected though, that the Proxy will be able to run on the NSLU2 when newer versions of the OpenSource VMs come out that support the missing features; or when Sun supports the ARM architecture. The architecture of the hardware / software that was used for the procedures mentioned in this section can be seen in the next figure:

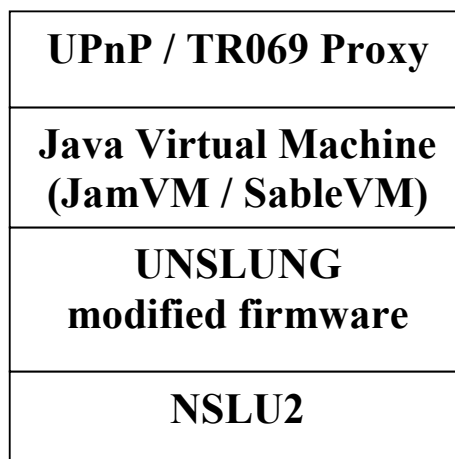


Figure 6: UPnP/TR069 Java Stack