Software Management of Heterogeneous Execution Platforms

André Bottaro[1,2], Levent Gürgen[3], Maxime Vincent[1,2], François-Gaël Ottogalli[1] and Stéphane Seyvoz[1,2]

[1] *Orange Labs*
38240 Meylan. FRANCE
{firstname.lastname}@orange-ftgroup.com

[2] *Grenoble University, LIG*
38041 Grenoble, Cedex 9, France
{firstname.lastname}@imag.fr

[3] *National Institute of Informatics*
101-8430, Tokyo, Japan
firstname@nii.ac.jp

Abstract

Time and cost efficient software maintenance is promoted to remotely manage devices distributed in the home network. In addition to the currently largely adopted monolithic software management, modular multi-tier software management needs are arising in order to respond to the requirements of various market actors using heterogeneous software platforms, e.g., Linux distributions, OSGi, .NET, MIDP. This paper defines a generic software management model, Genex, which provides a homogeneous view over these different types of platforms. The objective is to use a single generic platform that can manage several underlying heterogeneous platforms at a time. The paper also introduces our initial implementations of the model and presents some experimentation results.

Keywords: Device Management, Communication Protocols, Embedded Systems, Linux, OSGi, .NET.

1. Introduction

Device Management has recently become an important market need with the increasing number of devices in the home network. This brings important features for device/software providers to make time and cost efficient software maintenance. It covers several technical topics: software management, configuration, performance monitoring and diagnostics. Several groups are currently doing standardization efforts in these topics, e.g., the Broadband Forum (BBF), the Open Mobile Alliance (OMA), the International Engineering Task Force (IETF) and the Distributed Management Task Force (DMTF).

However, current protocols deal only with "monolithic" software management that concerns updating the device firmware as a whole – even if the update concerns a small patch to the current version. This is clearly not the optimal solution for the management of devices. Actually, today's devices, including embedded ones, are being enriched by more modular execution platforms such as Linux distributions, .NET, OSGi, Java MIDP. Modular software systems are adopted for their advanced features such as better flexibility, (re)configurability, reusability and interoperability management.

Nevertheless, device variety comes with the heterogeneity problem which has indeed become a reality in the home network [1]. Devices from different providers may support different software execution platforms. A management protocol for devices in a network should support all their underlying platforms.

Designing applications on heterogeneous devices made our group experience the advantages of software modularity on platforms such as OSGi and .NET. The lack of protocols in the software management domain now shifts the target of our contributions from API [2][3][4] to protocol design. This paper proposes Genex, a generic model that is mappable to existing home management protocols. The model consists of two parts: software units with state diagrams, and a set of generic management operations on the software.

Today, DMTF CIM [5], OMA SCOMO [6] and UPnP Device Management ongoing specifications [7] attempt to answer the technical requirements for such a protocol. The work described here is the baseline of our contributions to the specification of the UPnP Committee created recently. It has an impact on other standardization processes such as the OSGi Alliance [8] and BBF whose protocols [9] are extensible.

This paper is organized as follows. Section 2 introduces some use cases motivating the need for a generic software management model. Section 3 gives the definitions of various terms used in this paper and presents several existing modular software execution platforms. Section 4 defines Genex, which aims at providing a homogeneous view over those platforms. Section 5 presents our first implementation efforts.

Section 6 presents some recent related work and section 7 concludes the paper.

2. Device Software management challenge

Many execution platform technologies enable the management of modular software applications. It first enables the separate download of system parts, and thus the adaptation of device software to the environment requirements. The software provider maintains a set of modules without maintaining a set of complete systems.

Second, these technologies offer code sharing and isolation mechanisms between deployed software units to facilitate the re-use and evolution of units. The re-use by several applications allows resources saving.

Third, the management interface of these platforms enables the activation and the deactivation of running processes on the request of users and administrators. Service providers are interested in particular in remotely activating services required by clients.

Three main application types are envisioned:

- **Remote Management**: Service providers deploy, diagnose and repair software applications from management servers on the Internet.
- **Self-care**: The user manages their devices at home. Simplified tools monitor the evolution of networked applications and offer, for example, software management actions.
- **Network self-configuration**: Each device reacts to the software composition of the other in adapting its own composition and in possibly managing the others. Device software manageability is part of the autonomic concept of self-configuration [10].

Indeed, various software management models exist today. Linux distributions, .NET, OSGi and Java MIDP are analyzed in the next section.

Our contribution here is the design of a general software management model based on the analysis of available technologies. It leads to the specification of a platform-agnostic interface for software management.

3. Analysis of existing execution platforms

Advanced devices enable the management of fine-grained software units, we will focus on their common properties and particularities in this section.

3.1. Definitions

This section gives the definitions of terms used throughout this paper. First of all, we can define an **Execution Platform** as a runtime environment that can be customized and extended by applications via software units. It provides support for following software units and features:

- **Execution Unit** (EU): a functional unit that, once started, initiates processes to perform tasks or provide services, until that it is stopped. EUs are deployed by deployment units.
- **Deployment Unit** (DU): a self-contained binary unit that can be individually deployed on the execution platform. A DU may consist of resources such as library files and functional execution units.
- **Delivery Package** (DP): a binary unit delivering one or more deployment units to enable a consistent installation of an application. This package can be seen for instance as a zip file or an installer.
- **Dependencies:** the relations between software units, e.g., one DU can use a library provided by another DU, the execution of an EU may depend on the execution of another EU.
- **Metadata**: the description of software units, as well as their dependencies to other units.
- **Lifecycle management operations**: the tasks performed on various software units, e.g., download, install and start.
- **States**: software unit states change as a result of lifecycle management operations, e.g., downloaded, installed and started.
- **Events**: fired to notify a monitoring application of the changes on the state of software units.

3.2. Analysis

Many technologies enable to deploy and execute modular software applications. Next paragraphs give the relevance of mentioned technologies. Table 1 summarizes their characteristics.

Linux is now widespread in embedded devices of local networks. Redhat, Debian and Slackware define the main roots of Linux distributions. Most distributions derive from one of them. All the distributions seem to share the common definitions we mentioned above. Redhat RPM and Debian packages are examples of deployment units (DUs) of Linux platforms. Resources contained in these packages allow launching execution units (EUs) such as processes and daemons, with the help of init processes and RC scripts [13]. The Meta-package concept plays the role of delivery packages (DPs), which declare (and possibly contain) all interdependent packages. At some extent, the package is a minimal DP. Metadata is present in packages in order to describe their contents (e.g., name, version, description), as well as their dependencies. Packages are deployed via install tools such as rpm or apt-get. According to the installation process, they can have states such as installed, half-installed and resolved. Similarly, EUs can be in states such as active or inactive. It is possible to be notified of the activity

Table 1 Execution platform comparisons

Technology	Delivery Package	Deployment Unit	Execution Unit	Dependency	Actions	Events
OSGi	Deployment Package (mobile specification)	Bundle	Bundle	Bundle, Package, Service	Install, Start, Stop, Update, Uninstall	Installed, Starting, Resolved, Active, Uninstalling
Java MIDP	NA	Midlet Suite	Midlet	LIBlet (MIDP3 library)	StartApp, DestroyApp	NA
.NET	Windows installer	Assembly	Assembly	Assembly	Download, Load, Unload (AppDomain), Invoke	NA
Linux Debian	Meta-Package	Package	Processes (RC scripts in particular)	Package	PackageInstall, PackageUninstall, ServiceStart ServiceStop	Triggering Updates
OMA Scomo	Delivery Package	Deployment Component	NA	NA	Download, DownloadInstall, Install, InstallInactive, Update, Remove, Activate, Deactivate	Operational Results
DMTF CIM	Software Product	Software Element	Software Element	Dependency	Deploy, Install, Execute	(Ongoing work)

change of the processes thanks to the process table.

.NET provides a common language runtime (CLR), a multi-language support execution platform. It enables dynamic modular software management via its *assembly* concept. Assemblies are *DUs* for .NET applications. In the Microsoft implementation, there are two kinds of assemblies: libraries (dll) and *executables* (exe). They contain *metadata* giving information such as their name, version number and list of contained files. Metadata also declare *dependencies* to other assemblies. Dependency resolution is performed by the underlying platform. A shared assembly may be *installed* in a common repository named "Global Assembly Cache" (GAC). Private assemblies are *installed* in private folders of target applications. Alternatively, Windows Installers (i.e., *DPs*) can provide all necessary resources and assemblies to *deploy* a complete application. Executable assemblies can be *loaded* into "Application Domains" (ADs) and be *executed* inside of that domain via the method *ExecuteAssembly*. To uninstall an assembly, the corresponding AD should be *unloaded*. *Events* inform of the current *states* of assemblies and ADs (e.g., AssemblyLoad, DomainUnloaded, AssemblyResolve).

The **OSGi** technology [11] aims at being the dynamic module system for Java. It defines fine-grained code sharing and isolation mechanisms between *DUs* called bundles. Bundles are basically jar files providing libraries. A bundle containing a special class named Activator can be considered as an *EU*. Active bundles provide services to other *EUs* and applications to other actors (e.g., a graphical interface to users). *Dependency* information (e.g., exported packages, required services) can be declared in the bundle's *manifest* file, which also contains further information such as name and version. Every bundle can be atomically *installed*, *uninstalled*, *updated*, *started* and *stopped* thanks to the available methods, which cause bundles to transit to the corresponding *states* (e.g., Installed, Resolved, Uninstalled, Active). Bundles can be started only in the "Resolved" state, which indicates that required packages are effectively available. State changes fire the corresponding *events*.

Java **MIDP** [12] defines software units deployable on a constrained Java CLDC virtual machine. MIDlets are *execution units* for MIDP applications. They are *deployed* by MIDlet Suites which can contain one or several MIDlets. *Dependencies* are expressed in *metadata* files, namely, JAD files, of Suites and LIBlets. Suites can be *installed*, *updated*, *uninstalled* by application management software. MIDlets can then be *started* and *stopped* by the methods startApp() and destroyApp() respectively. The state changes (i.e., Active, Destroyed) are notified to the Application Manager.

The above analysis led us to envision a model covering the salient common characteristics. Next section introduces our model.

4. Genex: a generic execution platform

Genex aims to provide a generic model for modular software management, independent from the underlying execution platform technology. It defines common software units as well as actions to be performed on these units. The implementation of the model would do the mapping to the corresponding technology-specific actions on the corresponding software units.

4.1. Three software units

As a result of the analysis given in the preceding section, Genex identifies three software units: Delivery Package (DP), Deployment Unit (DU) and Execution Unit (EU). There is a hierarchical relationship between these units: A DP installs one or several DUs. A DU may contain several EUs. Genex allows the lifecycle

management of these units via various operations. Three actions are defined for DUs (see Figure 1).

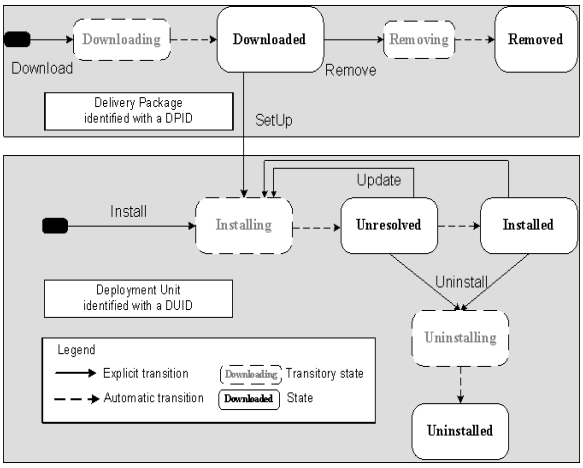

Figure 1 Delivery Package and Deployment Unit State diagrams

- **Install()** action installs a DU from a given URL and associates an identifier to it. During the installation, the DU enters ***installing*** state and when installation is complete, it enters the ***unresolved*** or the ***installed*** state**.** Installed state indicates that the DU is properly installed and its dependencies are resolved, thus the EUs present in the DU can be started**.** The ***unresolved*** state does not "guarantee" the proper functioning of the unit, for instance due to a missing dependency. Therefore, EUs are prevented to be started in this state, to avoid resolution errors at runtime. The transition between ***Unresolved*** and ***Installed*** states is automatic.
- **Update()** action takes the identifier of the DU to update and the URL of the new version as input arguments. State changes match the Install() ones.
- DUs are uninstalled via the **Uninstall()** action which takes the DU identifier as an input. DU enters the *uninstalled* state at the end of the action.

As mentioned above, DUs are self-contained units that can be installed individually. However for the consistent installation of a set of DUs, Delivery Packages can be used. Three actions are defined for Delivery Packages (see Figure 2):

- **Download()** action copies the DP to the device from a remote location identified by a URL. This action attributes an identifier to the delivery package and set its state to *Downloaded*.
- Once DP is *downloaded*, **Setup()** action on it installs all DUs contained in the DP. The effect is the same as invoking Install() on each DU contained in this DP.
- DP can be *removed* from the device by a **Remove()** action. This action does not uninstall the DUs installed by this DP. This action passes the DP to *Removed* state. This allows installers to be removed after their install job is done.

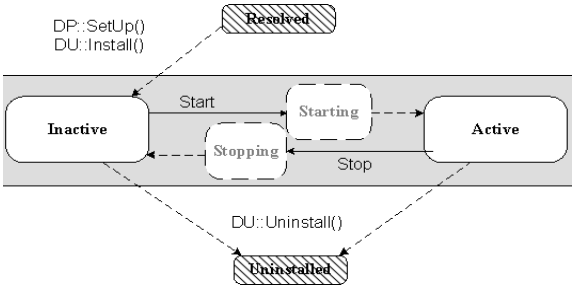

Figure 2 Execution Unit State Diagram

Once a given DU is *installed* and only in this state, its embedded library can be used by other units and its Execution Units can be activated. Two actions are defined for Execution Units (see Figure 2):

- **Start()** is used to activate an EU. The state of the EU changes to *active*. It becomes thus ready to be used by applications.
- **Stop()** stops the execution of an EU. Its state changes to *inactive*. When a DU is installed, contained EUs are also in the *inactive* state.

4.2. Dependency management

There are two types of dependencies. First, between units of distinct types: a DP may contain several DUs, and a DU, several EUs. For instance, the uninstallation of a DU implies the stop and the removal of contained EUs. Second, between units of the same type. Operations targeting a DP, a DU and an EU involve the same operation respectively on dependent DPs, DUs, and EUs.

Genex enables the management of the dependencies of the first type. Children of DPs and DUs are identified in the metadata of these units. Dependencies of the second type are left to the underlying device, if it is capable of it and if the manager wishes this automatic management. The manager has to handle dependencies and deploy individual units otherwise.

4.3. Event management

Genex allows the management of a device by several managers. These actors subscribe to the state change of software units in order to have an up-to-date view of the device. Notified events enable them to react and propose new applications. Execution platform technologies define events on state changes. Genex maps technology-specific events into the state changes of DPs, DUs and EUs and generates associated events.

The knowledge of ongoing operations is also important to managers. Every operation is performed asynchronously: An action initiates the operation and the identifier of the operation is immediately returned, then events associated to this identifier indicate the end of the operation to the service event subscribers. Resulting state changes are notified in this event.

4.4. Atomicity and isolation management

Transitory states indicate that software units are under software management operations. After that a management action, e.g., Install(), has been called without error on the management interface, the involved software units enter a transitory state before the output response is returned, e.g., Installing.

For isolation purpose, actions are prevented on the units that are in transitory states. These states can be considered as locks taken out on involved units.

For atomicity sake, if an operation fails, the involved software units should go back from transitory states to initial stable states. Operations are then performed on an all-or-nothing basis.

5. Implementations

5.1. Linux Debian environment

We have made an experiment of our high-level set of operations on the Ubuntu distribution. We reused the well-known APT middleware to download, remove, install, uninstall and update packages (and meta-packages) from repositories. APT manages the dependencies between packages on the distributions of the Linux Debian family. Init daemons and RC scripts [13] are matching our Execution Units with available start and stop operations.

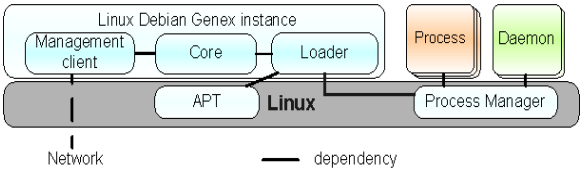

Figure 3 Linux Debian platform management

Software management operations are listened on the network by the Management client that calls the operations on the Core component. The latter turns these calls into platform-specific commands to the Loader, which uses APT and process management modules (see Figure 3).

5.2. Microsoft .NET platform

The implementation uses the .NET API in order to install, update and start assemblies. The download, setup and removal of Windows installers are part of our features. The download and removal of assemblies in the GAC are distinguished from the effective installation in Application Domains (AD). Assembly dependencies are managed by .NET, which loads necessary assemblies at runtime.

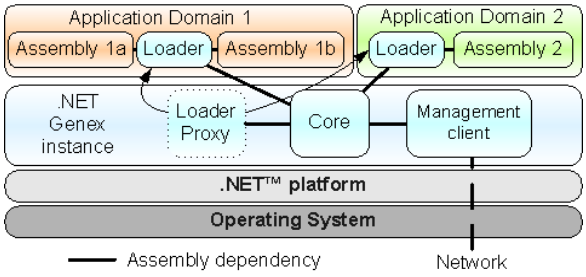

Figure 4 .NET platform management

However, two salient details prevent a natural mapping of the complete envisioned set of operations. First, an assembly can not be uninstalled without uninstalling the whole application domain. Second, executable assembly only present an invoke() method, which has to map Start() and Stop().

The implemented Core component (see Figure 4) communicates through .NET Remoting to the assembly loader instance loaded in each AD. .NET platform technical structure requires that assembly loading is made by an assembly of the managed AD.

5.3. OSGi platform

The OSGi API provides methods to install, update, uninstall, start and stop bundles. The Core component (see Figure 5) copes with the synchronous aspect of the methods to provide targeted asynchronous operations to the Management client.

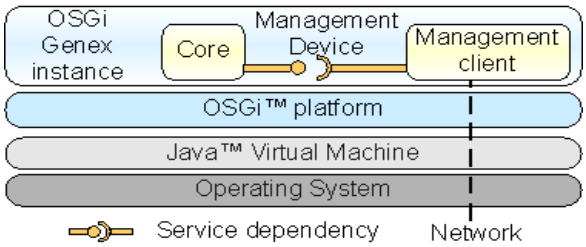

Figure 5 OSGi platform management

OSGi Bundle Repository (OBR) defines a syntax to declare bundle dependencies, which may be of distinct natures: packages, services but also user-defined dependency types. The existing OBR repositories and the Apache felix client enable to manage dependencies at install time. Dependency management at update and uninstall time remain a complex extension to realize

like on other execution platforms.

6. Related Work

The described work is closely related to SCOMO [6]. Indeed, the latter defines a platform-agnostic protocol to manage software units on devices. SCOMO defines two units: a Delivery Package and a Deployment Component. These units may be considered similar to Genex DP and DU.

However, SCOMO first does not catch the generality of the third entity we define here, the Execution Unit. Activation and deactivation are made at the DU level and therefore are ambiguous with the automatic resolution made by the device. Genex puts the activation and deactivation at the EU level and defines resolution mechanisms on DUs. Second, transitory states are missing in their model. Genex makes the internal locks visible in order to give an up-to-date view of the ongoing management operations in order to prevent concurrent operations to be requested. Third, SCOMO defines operations as a side-effect of the data node configuration in its data model whereas service operations like UPnP actions or TR-069 RPCs [9] are better at defining the semantics of an operation.

Another work for generic device management comes from the DMTF group [5]. It defines CIM, a Common Information Model, representing the managed environment. Its aim is to unify and extend the existing instrumentation and management standards (e.g., SNMP, DMI, CMIP). In particular, the CIM Application Management Model describes the information commonly required to manage software products and applications. Similar to our work, they define units such as Software Product, Software Feature, Software Element and Application System; as well as states such as deployable, installable, executable and running. This model is conceived to describe applications ranging from standalone desktop applications to sophisticated, multi-platform Internet-based ones. To our knowledge, it has not yet been implemented due to its complexity.

7. Conclusion

The standards of the home network make the dream of smart applications organizing distributed devices according to the user environment come true. However, protocol standards lack of a general model for the agnostic management of modular execution platforms.

We propose a model leveraging the modularity of advanced platforms, which can be distinguished from the state of the art by its generality. The key concept of this design is first the definition of three distinct units (*DP*, *DU*, *EU*), as well as state diagrams enabling to describe atomic and isolated operations, taking care of unit dependencies. It has been applied to widely spread platforms, Ubuntu Linux, .NET and OSGi, and is the baseline of our contributions to the UPnP Device Management Working Committee.

We are mainly investigating the integration of this general protocol in self-care tools addressing the dynamicity of pervasive networks like the Home network and the consistent deployment of distributed applications in local networks.

8. References

1 Sumi Helal, "Standards for service discovery and delivery", IEEE Pervasive Computing Journal, Volume 1, Issue 3, pp. 95-100, July-Sept. 2002

2 André Bottaro, Eric Simon, Stéphane Seyvoz, Anne Gérodolle, "Dynamic Web Services on a Home Service Platform", 22nd IEEE International Conference on Advanced Information Networking and Applications (AINA-08), Ginowan, Okinawa, Japan, March 2008

3 André Bottaro, Anne Gérodolle, Philippe Lalanda, "Pervasive Service Composition in the Home Network", 21st International IEEE Conference on Advanced Information Networking and Applications (AINA-07), Niagara Falls, Canada, May 2007

4 Levent Gürgen, Claudia Roncancio, Cyril Labbe, André Bottaro, Vincent Olive, "SStreaMWare: a service oriented middleware for heterogeneous sensor data management", 5th IEEE International Conference on Pervasive Services (ICPS'08), Sorrento, Italy, July 2008

5 Distributed Management Task Force (DMTF), "CIM Schema: Application Model Specification V2.13", September 2006

6 Open Mobile Alliance (OMA), "Software Component Management Object", Draft Version 1.0, June 2008

7 Jooyeol Lee, André Bottaro, Kai Hackbarth, Didier Donsez for the UPnP Forum, "Execution Platform Working Committee Charter", October 2007

8 Kiran B. Vedula, Jae Shin Lee, Hyun-Gyoo Yook, Peter Kriens, André Bottaro, "RFP 101 The OSGi platform as a UPnP device", OSGi Alliance, October 2007

9 Broadband Forum, "CPE WAN Management Protocol (Technical Report 69)", May 2004

10 Jeffrey O. Kephart, "Research challenges of autonomic computing", 27th IEEE Conference on Software engineering (ICSE'05), St. Louis, MO, USA, 2005

11 OSGi Alliance, "OSGi Service Platform Core Specification Release 4", October 2005

12 Java Community Process, "Mobile Information Device Profile for Java Micro Edition Version 3", February 2007

13 Yvan Royon, Stéphane Frénot, "A Survey of Unix Init Schemes", Technical Report, Inria RT-0338, June 2007

14 Apostolos E. Nikolaidis, Serafeim Papastefanos, Gregory A. Doumenis, George I. Stassinopoulos, Marios Polichronis K. Drakos, "Local and Remote Management Integration for Flexible Service Provisioning to the Home", IEEE Communications Magazine, October 2007