Robust Embedded Applications
OSGi ME Platforms

November, 25th, 2011

Olivier Patry & Fred Rivard, PhD
devel@is2t.com

Andre Bottaro, PhD
andre.bottaro@orange.com
Executive Summary

• OSGi is a reference for flexible component based platforms and standardization on the Home Market

• OSGi ME is an initiative from Orange Labs and IS2T to show
  - OSGi specification is simplified and targets embedded needs
  - Robustness is strengthened for openness to third party applications

• OSGi ME specification is released, IS2T delivered 1st implementation
  - Strengthened class sharing and isolation mechanisms
  - Hot plug-n-play mechanisms
  - Stale reference management
  - Transactions management
  - Compatible with SE basic libraries; eClasspath to enable OSGi developers to develop on most used Java SE classes
Why OSGi is a reference for the software embedded in the box
Smart Home

Seven application domains
- Security: An always-on system that reassure the user
- Energy: The optimization of energy and fluid consumption
- Comfort: A system assisting the user every day
- Health: The care of elderly people at home
- Wellness: A coach at Home
- Multimedia: content sharing
- Games: Networked community games and gadgets

A new world of applications will emerge from the variety of sensors, actuators, devices that become available
Opportunity: Catalyze the market through the share of a common technical infrastructure

• Leverage a common ICT infrastructure
  - Standard sensor networks
  - Advanced Home Gateway or a box dedicated to home automation apps
  - Remote Device/Network Management platform

• To lower the barriers and make the market emerge
  - Lower installation and maintenance costs of each service
  - Go beyond quadruple play, accelerate service delivery
  - Enlarge the Home Automation market
The Home ecosystem requires:

- a **standard** software environment
- **open** to third party applications
- with **agile** development techniques
- while remaining compatible with **embedded** constraints
  - Keep device cost low
  - Make robust applications
  - Create a dynamic market of applications with Home players:

- Applications designers (bundles & services)
- Platform administrators & application stores
- **OSGi ME providers** (SW part of the OSGi ME platform)
- Hardware providers (HW part of the OSGi ME platform)
OSGi model is the primary reference in Home standardization bodies

- **Broadband Forum**: Software lifecycle management is lead by OSGi concepts
- **HGI**: OSGi is the main software platform pushed in HGI specifications
- **UPnP Forum**: OSGi is the primary reference for Device Management
- **OSGi Alliance**: concepts and specifications
Why OSGi ME?
## Improvements of OSGi technology are expected on the Smart Home market

<table>
<thead>
<tr>
<th>OSGi ME technical improvement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptation to cost-effective devices.</td>
<td>OSGi ME runs on any Java ME CDC, SE, EE platform. OSGi ME minimum environment is closed to CLDC. EDEE = OSGi/EmbeddedDevice-1.0 Execution Environment</td>
</tr>
<tr>
<td>Strengthened code isolation</td>
<td>Several OSGi R4 methods are removed to avoid security checks.</td>
</tr>
<tr>
<td>Transactional guaranties</td>
<td>OSGi ME enables transactional bundle &amp; service lifecycle.</td>
</tr>
</tbody>
</table>

### Requirements beyond current OSGi ME

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware resource sharing and isolation on an open embedded software platform</td>
</tr>
<tr>
<td>OSGi is the technology reference on the market before .NET and Androïd but it addresses only code sharing and isolation.</td>
</tr>
<tr>
<td>Trusted pay-per-use services</td>
</tr>
<tr>
<td>Reliable usage counting</td>
</tr>
<tr>
<td>Device abstraction layer</td>
</tr>
<tr>
<td>Provide proxy interfaces for all type of devices, sensors, actuators at Home</td>
</tr>
</tbody>
</table>
Results so far

- JVM flash footprint ~200kB
- Footprint compatible for flash MCU mass market devices.

<table>
<thead>
<tr>
<th>Application components into OSGi™ bundles</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSGi™ R4 Platform</td>
</tr>
<tr>
<td>Java VM (&gt;Java ME CDC)</td>
</tr>
<tr>
<td>Linux</td>
</tr>
<tr>
<td>Home Automation Box prototype</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application components into OSGi™ bundles</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSGi ME Platform</td>
</tr>
<tr>
<td>JHAB (Java ME CLDC), baremetal</td>
</tr>
<tr>
<td>Home Automation Box prototype</td>
</tr>
</tbody>
</table>

One same application tested on two OSGi platforms running the same hardware.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>OSGi</th>
<th>OSGi ME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework footprint</td>
<td>~400KB</td>
<td>~40KB</td>
</tr>
<tr>
<td>JVM footprint</td>
<td>~2,000KB</td>
<td>~200KB</td>
</tr>
<tr>
<td>OS footprint</td>
<td>~11,000KB</td>
<td>11,000KB</td>
</tr>
<tr>
<td>Application footprint</td>
<td>~800KB</td>
<td>~350KB</td>
</tr>
<tr>
<td>Java RAM</td>
<td>~700KB</td>
<td>~550KB</td>
</tr>
<tr>
<td>Startup time (HW+SW)</td>
<td>33.0s</td>
<td>5.5s</td>
</tr>
</tbody>
</table>
Other results - Orange experiments

- Comparisons were made between IS2T Java vs JamVM on operations expected in home automation applications
  - smaller code is faster code!
- Strict CLDC APIs have been considered as too restrictive for minimum execution environment
  - OSGi developers ask for Java SE typical classes
  - Binary compatibility between all OSGi versions is required
⇒ IS2T provides now eClasspath, an additional set of Java CDC classes

<table>
<thead>
<tr>
<th>collections</th>
<th>Main J2SE collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>core</td>
<td>Core classes and interfaces</td>
</tr>
<tr>
<td>format</td>
<td>Text formatting classes</td>
</tr>
<tr>
<td>fs</td>
<td>File manipulation classes</td>
</tr>
<tr>
<td>locale</td>
<td>Localization related classes</td>
</tr>
<tr>
<td>math</td>
<td>Math BigDecimal and BigInteger</td>
</tr>
<tr>
<td>net</td>
<td>Network related classes</td>
</tr>
<tr>
<td>nio</td>
<td>New Input/Output J2SE profile</td>
</tr>
<tr>
<td>properties</td>
<td>J2SE properties</td>
</tr>
<tr>
<td>sql</td>
<td>SQL querying and database connection APIs</td>
</tr>
<tr>
<td>streams</td>
<td>Input/Output streams</td>
</tr>
<tr>
<td>timer</td>
<td>Timer and TimerTask</td>
</tr>
<tr>
<td>timezone</td>
<td>Timezone and daylight savings</td>
</tr>
<tr>
<td>url</td>
<td>URL</td>
</tr>
<tr>
<td>zip</td>
<td>ZIP inflated/deflater</td>
</tr>
</tbody>
</table>
OSGi ME in a nutshell
OSGi ME in a nutshell

• Keeping the core features of the OSGi technology...
  – fine-grained code sharing and isolation model
  – dynamic software management
  – requiring a full upward compatibility from OSGi ME to OSGi

• and being compliant with Java for “small” devices...
  – no user-defined class loaders
  – binary compatibility with Java CDC/SE APIs

• while simplifying OSGi technology for simpler needs...
  – remove unnecessary and semantically complex features

• and strengthen robustness.
  – no stale reference
  – transactional life cycle management
  – true code isolation
Fine-grained code sharing and isolation

- Modularity demands a strict model of what is guaranteed at runtime
  => A code change in a bundle has only an impact on bundles that declare code imports.

- Openness to third party applications demands an even stricter model
  => Distinct actors will only share object access through declared APIs

- OSGi foundation is a unique basis (vs Linux, Windows, .NET, Android,...)
  - A bundle declares code imports / code exports / code that remains private
  - OSGi framework guarantees bundle declarations
    - **Sharing**: a bundle will have visibility on APIs that are imported
    - **Isolation**: a bundle will never access private code and code that is not imported
  - Sharing is efficient: direct method call between bundles (no IPC)
R4 do not totally enforce sharing & isolation

- Any bundle can load any private class and instantiate it
  - BundleContext.getBundles() + Bundle.loadClass()
  - Class.forName()

- Any bundle has visibility on any implemented class signature
  - <service object>.getClass() gives the private name of classes implementing shared APIs

- Any bundle has visibility on any file or resource
  - BundleContext.getBundles + Bundle.getBundleContext().getDataFile()
  - BundleContext.getBundles + Bundle.getBundleContext().getResource()
  - BundleContext.getBundles + Bundle.getBundleContext().findEntries()

- Any bundle can register services for other bundles
  - BundleContext.getBundles()
    - + Bundle.getBundleContext().registerService()

⇒ OSGi requires Java 2 security to ensure isolation
ME ensures strict sharing & isolation

• Class lookup is refined
  - `Class.forName()` behavior depends on the context
  - `Bundle.loadClass` is banished since any `Bundle` object is accessible

• Private class signature are never visible to other bundles
  - `<service object>.getClass()` depends on the context

• The access to the bundle context of other bundles is banished
  - `Bundle.getBundleContext` is removed in OSGi ME
  - `BundleContext.getResource()` and `BundleContext.registerService()` thus really depend on a specific bundle context.

⇒ OSGi ME ensures a strict model for sharing & isolation
⇒ and does not require Java 2 security to do so
OSGi ME adapts Software Dynamics to business needs

• Different market requirements on binary code downloads
  - 1st case: No download due to certification, B.O.M., threats
  - 2nd case: Controlled downloads for only well-known (approved) services, with threat control, proprietary protocols via proprietary media, e.g., uart, spi, i2c, CAN, Ethernet, GSM, Zigbee, ...
  - 3rd case: Authorization of any download from any kind of sources

• Device Software Dynamics levels characterize how binary code is loaded into devices
  - DSD 0 = no download
  - DSD 1 = download through a controlled media
  - DSD 2 = no restriction
An OSGi technology compliant with Java for cost effective devices

- Java ME is Java most spread edition on embedded devices
  - Spread on mobile phones and part of M2M modules
  - OSGi specification is not compliant because it specifies how to implement sharing & isolation (class loaders)

- No (heavy and too flexible) user-defined class loaders
  - Although at the basis of OSGi code sharing and isolation model
  - And at the basis of OSGi dynamic software management features
  ⇒ Because they are not the only technical solution, OSGi ME specification does not mention class loaders

- Interfaces used at runtime must be declared at development time
  - Dynamic or optional imports are not compliant and not needed
  ⇒ OSGi ME reinforces OSGi class import declarative model

- Restricted Java APIs
  ⇒ OSGi ME is compliant with embedded Java environment
Stale reference management

• Problem
  - A service is a graph of objects
  - When a service is unregistered, service clients may still keep a reference on a “semantically private” object of the service graph (so-called stale references)
  - The overall OSGi platform gets unstable (memory links, ...)

• OSGi ME enables a robust dynamic service cooperation
  - Allow to avoid stale references
  - Users of services may control what to do when a service goes done
  - API to semantically build a service out of several cooperative objects
    ⇒ Explicitly attach objects to a “root” to build a service
    ⇒ A service = the attached objects to a root-object, which is registered
    ⇒ Throw a DeadObjectException on access to unregistered services or their objects
Transactional life cycle for robustness

- Installed
- Resolved
- Started
- Stopping
- Active
- Uninstalled
- Conditional execution
- Implicit transition
- Explicit transition
- Transactional context

Event sent synchronously to event listeners in a new transactional context (one for each even sent).
Demo 1: name space true isolation

• **Compile time**
  – check at link-time (SOAR) of the bundle visibilities & bindings
    • check for un-resolved import
    • manipulation of an un-exported type
  – check done within the desktop tooling ==> remove lots of costly runtime checks

• **Run time**
  – manipulation of exported types (i.e. “should be” only interfaces)
  – `getClass()` on an object from a “private-package” ==> throws SecurityException
  – `forName()` for an un-exported type ==> throws ClassNotFoundException

• **Debug time**
  – dedicated support for `toString()`/`getClass()` for JDWP debuggers
Demo 2: transactional life cycle

• What is a transaction?
  – Transactions semantic
  – Examples of transactions usage

• Application to transactional OSGi life cycle of bundles
  – One service from IData registered within the registry (getData()/setData())
  – 2 bundles make use of the service (under transactional context): BundleActivator.start() / stop()
  – Within the transaction, side effects are visible
  – On errors in the activator ==> rollback ==> the service remains unchanged, and not in an « in-between-state »
  – On success ==> commit, the updated service is visible to the rest of the application.
Demo 3: stale reference management

- **Usage of an object part of a service**
  - Case 1: wrong visibility: public field, export of a type
  - Case 2: correct visibility but poor implementation: the “public” objects of a service (a graph of objects) should have been attached to the service (Framework.attach())

- **Impact from stale reference usage**
  - The user of a service is not responsible for the “poorly buggy” service. The provider of the concrete implementation must be blamed!
  - Actions to be taken by the administrator of the device
    - Detection of the stale references, and advocate the provider.
    - At some point, the administrator may remove stale references

- **Usage of a service**
  - A service may go done, fail, etc.
  - Need to make provision for DeadObjectException
Demo 4: good design practice

- **Bundle Dependencies**
  - Leaf of the dependencies tree == implementations bundles
  - root and “core” of the tree == interfaces bundles
  - Internal framework special case:
    - Leaves may be an implementation tree, with abstract “default” classes
    - typical usage: a framework implementation. The framework exposes its interfaces, but implementation bundles uses “internal” exported bundles (*x*-internal typical convention within Eclipse)
Roadmap and next steps

• Orange & IS2T specification is available at http://www.e-s-r.net/en/010osgime.php

• IS2T provides OSGi ME implementations for several hardware

• Next steps in Open The Box French national project
  - Specify and insert resource management into OSGi ME framework specification for services: CPU, memory, IO bandwidth, ...
  - Trust on pay-per-use services
  - Implement simplified Java 2 Security mechanisms
  - Strengthen protection on intellectual property
Thanks!

Olivier Patry & Fred Rivard, PhD
devel@is2t.com

Andre Bottaro, PhD
andre.bottaro@orange.com
DSD0: Device Software Dynamics 0

- The code is loaded using some probe, e.g., JTAG, or bootloaders
  - The device needs to be “switched off” to download the whole application
  - No runtime download

- A closed system
  - OSGi ME to design a component based application
  - The whole application code is known at link-time
  - Full off-board linking is feasible
DSD1: Device Software Dynamics 1

• **Code loading under strict control**
  - The media by which the code gets downloaded to the device is controlled by some technical means, most probably protected by some proprietary protocol.

• **A controlled system**
  - OSGi ME to design a component based application
  - The application can download bundles at runtime
  - Pre-linking and off-board pre-analysis is feasible
  - Adding and/or Updating are feasible
DSD2: Device Software Dynamics 2

• **Free downloads**
  - The device has to embed all the necessary protections.

• **A fully open system**
  • OSGi ME to design a component based application
  • The application can download any bundle at runtime
  • **All linking** is done on-board
  • Adding, updating, uninstalling feasible without constraints