RFP 52 - MEG Application Model

Draft

21 Pages

Abstract

Requirements for the MEG application model.

0 Document Information

0.1 Table of Contents

All Page Within This Box

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
V 0.1	Jan 12 2004	Vadim Draluk, vdraluk@motorola.com
V 0.2	Jan 23 2004	Vadim Draluk, vdraluk@motorola.com 1. Terminology changed to one of "units" 2. Use cases added for UE pause/restart and termination
V 0.3	Jan 30 2004	Vadim Draluk, vdraluk@motorola.com 1. Editorial changes from Nokia accommodated 2. Problem description section added
V 0.4	Feb 23 2004	Harry Prabandham, harryp@motorola.com 1. Editorial changes from Nokia and Prosyst accommodated.

All Page Within This Box

Revision	Date	Comments
V 0.5	March 1 2004	Harry Prabandham, harryp@motorola.com 1. Merge MIDP Requirements into this document 2. Added aways active vs. on demand bundle/service activation as an explicit requirement 3. Added explicit requirement to minimize the classloaders to conserve RAM.
V 0.6	March 6 2004	Harry Prabandham, harryp@motorola.com 1. Incorporated comments discussed in conference call 03/02 with Nokia, Prosyst and Peter Kriens. 2. Incorporated comments by Erkki Rysa on the merged document.
V 0.7	April 11 2004	Harry Prabandham, harryp@motorola.com 1. Incorporated comments from Nokia (Robert Fajta). 2. Incorporated comments from Nokia (Aleksi Uotila). 3. Incorporated comments from Siemens (Calinel Pasteanu)
V 0.75	April 13 2004	Harry Prabandham, harryp@motorola.com 1. Incorporate feedback from Nokia (Robert Fajta).
V 0.9	23 Apr. 2004	BJ Hargrave, hargrave@us.ibm.com Formatted for external review.

1 Introduction

This document is part of overall effort of the *Mobile Expert Group* (MEG) under OSGi to come up with use cases and requirements for operational management of mobile device platforms based on CDC configuration of J2ME and OSGi. The effort is subdivided into several *Work Streams* (WS) as follows:

- Application model
- Device Management
- Policies
- Deployment

It also encompasses two horizontal efforts, which manifest themselves in all vertical aspects of the platform:

- Security
- Non-functional requirements and features

This document is devoted to consideration of use cases and requirements pertaining to the Application Model (AM). It addresses the issue of new Units of Execution (UEs) and other artifacts necessary (or simply convenient) for support of a simple mobile-device-specific application development

2 Application Domain

Originally OSGi has been designed as a very generic and horizontal services framework, with its design centered on flexibility of service deployment and consumption. Aiming at a wide variety of application domains, it could not (and should not) have addressed in detail the question of how to make application development more straightforward and convenient. This task belongs to the vertical domains, now expanding their use of OSGi.

Mobile devices area is the latest newcomer to the OSGi world. The only well-defined application model today for such devices is MIDP, a very important but limited platform, specifically designed to make Java available for small applications development, but unsuitable for using Java to build re-usable programmatic services. This document lists use cases and requirements driving introduction of new, mobile-specific artifacts taking full advantage of

OSGi framework's flexibility, while building on and expanding the lifecycle features of MIDP applications (MIDlets).

This document also covers use cases and requirements for an environment, which allows the Mobile Information Device Profile (MIDP) applications to run on top of the OSGi framework. The effort is driven by the idea not to merely provide another execution environment, but to extend the MIDP applications' functionality by providing a new level of interaction with the outer world, like usage of OSGi services, reuse of already deployed modules, etc.

The MIDP application model has gained significant popularity in the mobile world. Support for Java/MIDP has become a standard feature of modern mobile phones, and downloadable games and applications, provided by service operators, are generating more and more revenue. So building a next generation mobile phone is unthinkable without MIDP.

This document focuses on requirements for a new type of service delivery and execution environment for MIDP applications based on an OSGi framework. Such an environment must preserve the existing application model, but also extend the MIDP applications (MIDlets) lifecycle and features, by utilizing the OSGi framework's flexibility.

From MIDP developer's perspective:

- Existing investment in MIDP application has to be preserved.
- Expose some of the dynamic features of OSGi to MIDlets to enable sharing common classes and libraries.

2.1 Terminology

UST – Unit of Staging, a collection of OSGi artifacts used by the server side management system to stage an application on the server for the sole purpose of making it available for device for download/installation. This artifact may not necessarily be known to the management system on the device.

An OSGi artifact is a standard OSGi bundle augmented with additional information necessary for the management system defined by the MEG framework.

UDL – Unit of Delivery, a collection of OSGi artifacts used by the management system to deliver an application to the device, but not necessarily known to OSGi framework.

UDP – Unit of Deployment, an OSGi artifact associated with the minimal separately deployable entity. This is currently equivalent OSGi bundle with additional descriptor (definition below) information.

UE – Unit of Execution, an OSGi artifact associated with its own execution context.

Descriptor – a file associated with one of the units defined above and containing information necessary for that unit's processing by the MEG framework.

Note: The above terminology has been defined to reduce the overloaded terminology that application, service, bundle etc. brings to the discussion. In the RFC stage, these may be mapped into concrete file formats for inter-operability.

All Page Within This Box

3 Problem Description

This document is intended to address several issues that are specific to an OSGi-based application model for mobile devices:

- UDLs and UDPs well-suited for all modes of application delivery to the device, both connected and un-connected
- Lifecycle and interfaces supporting it for UEs
- Role of mobile-specific events and notifications in UE lifecycle
- Generic model for events/notifications
- Dependencies between various units, including version dependencies

In the case of MIDP, the most crippling aspects of the MIDP 1.0/2.0 specifications seems to be:

- Inability to reuse/share already deployed modules between MIDlets. For example, Palm platform enables separate deployment of libraries and applications that will use them, which minimizes the redundant code. Oppositely, the MIDP environment only shares the code, which has been bundled within the same unit of deployment.
- Inability to communicate with other applications. Though different MIDlets from a single MIDlet suite can share code and data (because they are loaded by the same class loader) it is not possible to do so with MIDlets from different MIDlet suites. This parries the possibility to build sophisticated multi-application models, i.e. launching an e-mail client from the browser, or vice versa (having in mind a more complex case than MIME-handling here).

By employing the power of OSGi platform we will be able to:

- Support both OTA provisioning of MIDlet suites and OSGi bundle-based;
- Run MIDlets directly on the OSGi framework (i.e. in the same VM);
- Allowing MIDlets to access already registered OSGi services, opening them to the outer world. This also makes it possible to build an inter-application communication;
- Allowing support for additional APIs provided as; downloadable/updateable bundles whenever possible.
- Support MIDP 2.0 security and allow alternative, configurable security policies.

4 Use Cases

4.1 Lifecycle Use Cases

This set of use cases describes situations encountered by mobile application developers, as they create new software for mobile devices based on OSGi framework and services. While packaging their applications as USTs, UDLs or UDPs, these developers have to use more fine-grained UEs to define the requisite functionality. UEs are similar to application entities known from platforms designed for different environments: Servlets for Web application containers, applets for browser-based applications, midlets for CLDC/MIDP-based mobile environments etc.

No specific use cases are considered for USTs since they are by definition invisible to the device. UDPs will be considered to be identical with OSGi bundles as currently defined in the 3.0 version of the OSGi framework, though their manifest data may need to have to be upgraded with some additional MEG-specific information.

4.1.1 Full application delivery to the device

An application is packaged by the carrier or enterprise IT to contain multiple UDPs, possibly originating from various vendors with different level of trust. It is then delivered to the device as a UDL on a medium not connected to the server (memory stick, for example), and has to be installed without the device being connected to the net, and hence the server-based management server.

Such a UDL then has to contain all the information necessary for installation on the device, including both software components (services and packages) required by the application, and requisite resources, data files, policies and configuration parameters. MEG framework should provision these configuration parameters into the Device Management Tree and remove these from the Device Management Tree when UDL is removed.

4.1.2 UE activation and de-activation

A piece of functionality associated with an execution context is created by a developer. Its lifecycle is shorter than one of the entire OSGi bundle, so that it can be explicitly activated and de-activated within the bundle's lifecycle.

A UDP can contain multiple UEs, which can be activated and de-activated independently. UE activation is performed by the UE's environment which manages its execution state.

Starting the bundle creates a common context for UE execution, stopping the bundle removes this context and makes starting UEs impossible. Under this usage pattern, start() and stop() methods for the bundle are similar to invocation of DllMain() in Windows dynamic libraries.

4.1.3 Passing runtime parameters to a starting UE

A UE takes some input parameters which affect its execution. In case it's started, say, through its icon or corresponding menu item being selected, these input parameters have to be externally defined, for example in the UE descriptor, so that they are associated with this particular UE.

4.1.4 UE suspend/resume

While a UE is executing, high-priority event can occur that may require scaling down of the UE's activities. This event can be for example resource shortage or an event that needs urgent attention of the user. In this case the UE is suspended and can be resumed later.

A suspended UE is not executing its designated function, but can retain some limited ability to process critical events. However, the ability of processing an event is a property of the UE itself. Events delivered to a suspended UE may be filtered so that only critical events are delivered.

4.1.5 UE pause/restart

A high-priority event occurs requiring complete temporary stop of all UE's activities, so that the UE is paused. A paused UE does not execute its designated function, not is it processing any external events, so that all events occurring between *pause* and the subsequent *restart* are lost and left un-processed by the UE. Stopping a UE destroys the context, while the pausing a UE saves the context (need not be persistent) that can be reused for a subsequent restart.

4.1.6 UE locking/unlocking

Locking of a UE prevents it from being started until it is unlocked. To be locked, a UE should not be running at the time.

UE can be locked for different reasons. For example, some functionality can be locked when the device is handed over to a child for temporary use. This can be achieved by corresponding policies in environments that feature explicit multi-user support, but requires separate interface in single-user systems.

Another use case is one of an operator choosing to lock an application for business reasons. The locking does not affect the application that is already running.

4.1.7 UE prioritization

Since UEs may be associated with execution threads, they can run in parallel, and with different priorities. Such priorities will be optionally set in the UE descriptor, thus associated with a particular UE. Operators can change the UE priority dynamically.

4.1.8 UE launch by another UE

Suppose an email client bundle is developed. It can have two distinct UEs: one listing all the messages in a mailbox at once, with minimal information about each message (such as "From", "Subject" and timestamp), another viewing the specific message in all detail.

It makes sense to implement the two as separate UEs for modularization purposes. On the other hand, the listing view UE has to be able to invoke/launch the detail view UE directly. One UE does not have to be in the same UDP to launch the other UE.

4.1.9 UE launch due to an event

A UE can be launched in reaction to number of events, such as timer (scheduled execution of synchronization procedures), arrival of a special SMS message requesting its execution from outside the device etc. Startup parameters can be either carried by the event object (like in case of SMS) or pre-configured in a corresponding UE descriptor. UE may also be terminated/suspended/resumed/paused/restored, maybe locked/unlocked by an event (if permissions allow it). For example, an event call completed event has finished could result in the call handling UE to stop.

This use case does not imply direct use of JSR 120 (Wireless Messaging API) by the application when SMS is processed: instead, they will be used by the MEG framework, which, recognizing that a particular message type is expected to result in UE execution, will produce a generic event of an appropriate type.

4.1.10 UE launch for content type processing

One specific type of event likely to activate a UE is triggered by an application which recognizes a certain MIME type as one to be exclusively processed. Examples of such applications include browser and file manager.

The data has to be passed to the UE for processing, without being handled by the browser itself. As in previous use case, the MEG framework will use APIs aligned with JSR 211 (Content Handler API), not the UE itself.

4.1.11 UE associated with GUI elements

A UE may or may not have a graphical user interface. But in case it can be launched from the device's GUI launcher (say for example, main menu), it has to be able to be associated with GUI elements representing this UE in the execution environment. To support such an association, the UE's descriptor optionally carries entities like icons for various UI environments, such as "desktop", menus etc. These all icons, names and descriptions must be able to be localizable so that one UE may contain resources for multiple locales. In addition the UE must have some sort of "group" and "group name" that allows it to be installed on e.g. on certain folder or directory within the device "shell" user interface. These directories can be e.g. generic "games", "applications", "connectivity", "office" or very specific "FooBar Company Utils".

4.1.12 UE enumeration

For monitoring purposes, both locally and remotely, UEs have to be enumerated. Enumerating globally implies enumerating UEs at the MEG framework level. Enumerating locally implies Enumerating UEs at the UDP level. Enumerating remotely is same as the globally, but it happens outside the device via some remote management agent.

4.1.13 Control of maximum number of active UEs allowed

Some UEs can have certain requirements for resource use, either exclusive or incremental, so maximum number of concurrently running UEs will have to be restricted at times. This can apply to both instances of a particular UE, and members of some UE groupings. These groupings can be defined by declaring UE's association with a particular group in its descriptor.

4.1.14 UE termination

Some UEs may have to be able to terminate themselves rather than expecting the MEG framework to do it. One example is a UE launched due to an event such as SIM card insert. The UE does not feature any GUI, but performs necessary SIM-based provisioning operations, and then terminates itself. Similarly, a UE can decide to suspend, lock or resume itself. For example, a UE initiates a file download, goes into a suspend state and goes into a suspended state, and at a later point, it comes back to an active/running state.

4.1.15 Resource clean-up at UE suspension and termination

When a UE is terminated or suspended, the MEG framework has to be able to reclaim resources consumed by it. This will happen temporarily in case of suspension, so that the resources will be re-assigned to the UE when resuming its execution, or permanently in case of its termination

4.2 Event and Notification Use Cases

4.2.1 Notification infrastructure

UEs running within Mobile devices are typically designed to react to a lot of asynchronous events happening within the device as the device interacts with the external world. Some examples of such asynchronous events are – Radio Network Signal Strength change, Incoming SMS/MMS message, Incoming Call, Insertion/Removal of an accessory. Although not all UEs are expected to use these events, a common infrastructure for such an event notification infrastructure is necessary for the UEs to interoperate. Here are some of the characteristics of the event infrastructure.

4.2.1.1 Event classification by type

When a generic MEG framework event is generated, its type has to be conveyed to the consumer for high-level processing and dispatching. This type can be just a string, allowing the event consumer to determine whether the event was a timer going off due to a pre-scheduled activity, an external message arriving, or an accessory attached to the device.

4.2.1.2 Wildcards in event types

In some cases similar actions (e.g launch of the same UE) have to occur when the event type corresponds to a certain pattern. One example is use of the Device Management Tree URI of the node as part of the event type reflecting changes made to the sub-tree. In this case using wildcard at the end of the event type string would enable uniform processing of all changes to a certain sub-tree.

4.2.1.3 Event data passing

While the event type reflects at the high level the nature of the event, its details are very specific to the event source: for the timer it would be a timestamp, for a message the details would contain its body and headers, for a DMT change event the exact URI of the affected node(s) as well as type of change etc.

These details have to be passed to the processing application in an agreed and generic way, allowing the detail processing to occur

4.2.1.4 Processing events between Java and native applications

Some of the events meaningful for the Java-based MEG framework originate in the native code, and vice versa. An example of the former is likely to be a physical event, such as SIM card insert or attachment of an accessory. The latter may happen if a Java-based DM facility sends notification of a change in the browser configuration, and a native browser application has to adjust its state accordingly.

4.2.1.5 Multiple event processing

The event processing mechanism has to have the ability to associate the same code with multiple events. For example, both SIM card insert and SIM card data change event have to result in invocation of the SIM-to-device synchronization code. Another example is an application reacting to changes in several sub-trees of the DMT

4.2.2 Specific notifications

4.2.2.1 Standard OSGi framework events

OSGi framework already has number of its own events defined, such as ServiceEvent, BundleEvent and FrameworkEvent. For uniformity of processing, these have to be mapped into generic event types, generated by listeners built into the OSGi framework. Such an approach is useful if the same code has to process multiple events, some of which are original OSGi framework events

4.2.2.2 MEG UEs lifecycle events

Similar to standard OSGi framework events, UEs lifecycle event type has to be introduced to associate pre-defined activities with UE's start, stop, suspend etc.

4.2.2.3 Logging event

Logging events are useful for allowing external alerts to be generated when certain error messages are logged.

4.2.2.4 SIM card events

Some devices allow replacement of the SIM card without a re-boot of the device. In this case SIM removal event is important to activate code disabling some basic functionality, whereas SIM insert events re-enables it, as well as launches OMA-standard SIM-based provisioning.

4.2.2.5 SIM card data change events

Some configuration data may reside on the SIM card, and be provisioned through SMS transparently to the MEG application framework. If the device's operating system is capable of issuing some kind of notification when this happens, SIM data change event can be issued to enable the MEG framework to perform the processing necessary, such as re-reading the data from the SIM card.

4.2.2.6 Configuration data change events

Applications affected by configuration data change have to be able to receive notification of such an event. Configuration data may be application-specific (browser setup, email parameters and such) or system-wide (such as GUI settings)

4.2.2.7 Telephony/Call Handling events

Actions have to be taken when a phone call comes in or is terminated. For example, some applications have to be suspended when a call is answered. Some applications may undergo visual changes based on the generated UI event when call is in progress. Other operations (example, data networking capabilities) of the application may have to be blocked or suspended for the duration of the call. Such events provide a generic mechanism to provide the necessary capabilities.

4.2.2.8 Timer/scheduling events

UEs carrying maintenance functionality can be scheduled for execution. One such function is PIM or email data synchronization. Another example would be some periodic resource clean-up. Similarly, as an example, this timer event could be used for stock exchange ticker UE, that can be started at 9 am automatically when the stock market opens, and closed at 5 pm when the stock market closes.

All Page Within This Box

4.2.2.9 Accessory attached/detached events

Attachment of an accessory may lead to visual or lifecycle changes in applications associated with them, so these applications have to be notified of such events. For example, an application exclusively handling an accessory has to be either suspended or stopped once the accessory is detached, and resumed or started when it is attached.

4.2.2.10 Resources low events

Resource-low conditions may affect some UEs' lifecycle. Resource-intensive UEs will have to be suspended or stopped, based on resource-low event notifications.

Note: One possible approach is to have a generic resource low event and provide additional data to identify the type of the resource that is low.

4.2.2.11 Messaging events

Message processing can also affect UEs' or other components' lifecycles. For example, WAP Service Indication/Load message typically causes the browser to be launched. A WAP push message can re-activate a SyncML DM agent. Some SMS messages, based on the port number specified, may result in bundle or Midlet provisioning agent to start.

4.3 Dependencies Use Cases

4.3.1 Dependencies of UEs on services

UEs can require certain services in order to be executed. For package dependency resolution, OSGi specified dependency resolution rules seem sufficient. For services, there are 2 types of dependencies (not package) – strong dependency and weak dependency. A strong dependency implies that such dependencies have to be listed in descriptor in order to be resolvable at the bundle's start time. Whereas a weak dependency can be resolved at runtime dynamically and the application can take care of the cases when the service is not available. A weak dependency does not have to be declared in the descriptor. The MEG framework based on the meta information provided in the descriptors, automatically figures out and starts the bundles and provides the service reference to the caller. In the case of weak dependency, the UE (or another service) must be capable of handling the condition when the service object is not available. As an example, UE could decide not to expose the functionality to the end-user if it is not available, or in other cases, UE could wait until the service object becomes available.

Strong dependency is equivalent to the case when a unix based application is linked with a set of dynamic libraries. If these libraries are not available, the application cannot be started. The weak dependency on the service is equivalent to the application calling dlopen() directly. If the library is not present, the application must be designed to handle such a situation.

4.3.2 UE version dependencies

UEs can have several kinds of version dependencies on the service that they depend on. UE may also depend on specific versions of services they use and the MEG framework should provide help in resolving these dependencies.

4.3.3 Native code dependencies

Any Java components may have dependencies on native code. Currently OSGi handles code dependencies based on hardware and OS platforms. Additional level on dependency tracking can be associated with native libraries' versions. In order to track those, library versions have to be specified in the descriptors

4.3.4 Physical resource dependencies

UDPs and UEs can have dependencies on particular physical resources on the device, such as screen resolution, accessory availability, radio interface etc. In order to support automatic recognition of code-to-device compatibility, device resource information has to be available, either through an API or through a descriptor. Similar information describing components' requirements has to be defined in these components' descriptors.

TBD: How this relates to UAProf? (http://www.openmobilealliance.com/tech/profiles/ccppschema-20030226.html)

4.3.5 Logical Resource Dependencies

Resource dependencies may be described by the descriptor in the UDP as a set of logical resources. A UDP may export and import these logical resources in the same way as Java packages. These logical resource dependencies are resolved and made available to UEs during the execution. An example of this is a UDP that exports not just Java classes but also localized images, resource bundles containing translations etc.

4.4 MIDlet Use Cases

4.4.1 Run existing MIDlets

To leverage existing investment in MIDP, and to avoid mobile developer fragmentation, it should be possible to run MIDlets within the MEG framework. From the end user perspective, running MIDlets or other UEs should be indistinguishable (user doesn't really care). The MEG framework around UE should be applicable to MIDlets so that they can be managed exactly the same way.

4.4.2 Share common Class Libraries across MIDlets

Currently, MIDlets have no way of sharing the common classes. Allowing MIDlet suites to share classes will lead to smaller a MIDlet suites and result in a faster application provisioning response times.

4.4.3 Add/Update/Remove JSRs or Carrier specific Licensee Open Classes

Currently MIDlets running on CLDC are limited to the set of JSRs that are provided by the device manufacturer at the factory time or the set of class libraries that are provisioned as a part of the MIDlet suite. In addition, there is no way of sharing of the common classes across MIDlet suites. By leveraging the power of the OSGi platform, carriers and operators want to make new JSRs and Class Libraries available to the MIDlets dynamically over the air (or via some tethered mechanism like USB, Serial etc.). Carriers also want to upgrade the version of the OSCLs that were shipped on the device to a newer version and have these available to MIDlets. These classes are immediately available to all the MIDlets to use, no restart of the device would be necessary.

Although, there is nothing in the CLDC or MIDP specification that would limit device implementations to provide more flexibility, any such extension would be deemed non-standard and will not result in a Midlet jar that would be interoperable.

4.5 Application Containers

MEG architecture must enable creation of multiple application containers (besides MIDP – which has already been discussed in the previous section) to avoid the fragmentation of mobile java programmers. It should make the execution of the existing non OSGi compliant applications possible within in the framework. A few examples of such application containers are: DoJa (DoCoMo Java) application container, VFX/VSCL container (Vodaphone Java Extension), Xlet container etc. In order to run an application the user need a special container, designed to nest the specific type of non OSGi compliant applications. These applications are treated as UEs by the MEG framework.

MEG architecture must support a pluggable architecture to support additional application containers. This approach will prevent developer fragmentation and make the OSGi relevant for supporting additional non-OSGi compliant applications. Such application containers can be preinstalled on the device but it should be possible to install/update/remove different kind of containers as necessary. The container provides the execution environment to the nested application maps the differences between the lifecycle model of OSGi/MEG framework and the nested application. The application container also maps the differences between security models. The running nested application within the containers can access the services, exported packages and extension APIs within the OSGi/MEG framework. The nested application may also provide service or export packages as necessary.

From the users perspective there should be no difference between OSGi/MEG compliant applications and the application running within the application containers. All these are treated as UEs by the MEG framework.

5 Requirements

5.1 Lifecycle Requirements

REQ-APP-01-01. A new UE, tentatively called *UE*, MUST be introduced

REQ-APP-01-02. UEs MUST be contained in UDPs.

REQ-APP-01-03. UEs MUST have descriptors, formatted as XML applications, contained in the same UDP as the UE.

Note: This descriptor is different from MIDP defined JAD/manifest syntax and OMA has defined OMA downloading descriptor. This descriptor is a part of the UDP and is not used by the download agents, but used to define meta information needed by the MEG framework for operational management.

REQ-APP-01-04. UE MUST support start() and stop() functions in its interface

REQ-APP-01-05. UE MUST be able to start and stop only after the containing UDP is started

REQ-APP-01-06. UE MUST support suspend() and resume() functions in its interface

REQ-APP-01-07. UE MAY support lock() and unlock() functions in its interface

All Page Within This Box

REQ-APP-01-08. UE MUST be able to accept startup parameters from its descriptor

REQ-APP-01-09. UE MAY assume priority relative to other UEs in the MEG framework as specified by its descriptor

REQ-APP-01-10. UE MUST be able to start another UE, and pass startup parameters to it

REQ-APP-01-11. The MEG framework MUST be able to release resources when the UE is stopped or suspended

REQ-APP-01-12. The MEG framework MUST be able to enumerate all UEs contained within a UDP

REQ-APP-01-13. The MEG framework MUST be able to retrieve UE's state (starting, started, stopping, stopped etc.) in all cases (including the case when the UE is locked).

REQ-APP-01-14A. The MEG framework MUST be able to control the maximum number of instances of the same UE active in the system at the same time, as specified by the UE specific descriptor.

REQ-APP-01-14B. The MEG framework MUST be able to control the overall maximum number of all UE instances active in the system at the same time.

REQ-APP-01-15. The MEG framework MUST be able to activate a UE in response to an incoming event, as defined by the UE's descriptor, and pass the data of this event to the UE

REQ-APP-01-16. The MEG framework MAY be able to activate a UE in response to an incoming MIME type, as defined by the UE's descriptor, and pass the data to it for processing

REQ-APP-01-17. UE lifecycle must be compatible with OMA DRM. For example, the starting and stopping of UEs must support OMA DRM protected content specification.

5.2 Event and Notification Requirements

In this section, we just define the need for the event types and do not specify mandatory/optional status for these event types. The mandatory/optional status for these events will be discussed and agreed to during the RFC stage.

REQ-APP-02-01. A generic event model MUST be supported by MEG to address impact typical mobile device event may have on lifecycle and management of the applications

REQ-APP-02-02. Events MUST feature a type, represented by a string

REQ-APP-02-03. Applications MUST be able to subscribe to one or to multiple event types

REQ-APP-02-04. Applications MUST be able to subscribe to events based on type patterns, using wildcards

REQ-APP-02-05. Events MUST carry data objects specific to event type

REQ-APP-02-06. Native applications MUST be capable of generating events which will be processed by Java applicatios

REQ-APP-02-07. Native applications MUST be capable processing events generated by Java applications

REQ-APP-02-08. Mechanism for marshalling/un-marshalling event objects between Java and native applications MUST be supported

REQ-APP-02-09. Event type corresponding to OSGi framework's FrameworkEvent MUST be defined

REQ-APP-02-10. Event type corresponding to OSGi framework's BundleEvent MUST be defined

REQ-APP-02-11. Event type corresponding to OSGi framework's ServiceEvent MUST be defined

REQ-APP-02-12. Event type reflecting UE's state changes (both lock state and lifecycle state) MUST be defined

REQ-APP-02-13. Event type reflecting record logging MUST be defined

REQ-APP-02-14. Event type reflecting framework boot (events that indicate Framework Initialization Begin and Framework Initialization Complete) MUST be defined

REQ-APP-02-15. Event type reflecting SIM card insert/removal MUST be defined

REQ-APP-02-16. Event type reflecting SIM card data changes MUST be defined

REQ-APP-02-17. Event type reflecting configuration information (both device's and UE's) changes MUST be defined

REQ-APP-02-18. Event type reflecting telephony call initiation/termination MUST be defined

REQ-APP-02-19. Event type reflecting WAP push message arrival MUST be defined, and contain the application ID as part of the type, thus enabling both application-specific and generic subscriptions

REQ-APP-02-20. Event type reflecting SMS/MMS/EMS/EMAIL message arrival MUST be defined, and contain the port number as part of the type, thus enabling both port-specific and generic subscriptions

REQ-APP-02-21. Event type reflecting timer-based scheduling MUST be defined

REQ-APP-02-22. Event type reflecting RAM-low condition MUST be defined

REQ-APP-02-23. Event type reflecting file-system-memory-low condition MUST be defined

REQ-APP-02-24. Event type reflecting battery-low condition MUST be defined

REQ-APP-02-25. Event type accessory attached/removed condition MUST be defined

REQ-APP-02-26. Event type reflecting extra memory card insertion/removal MUST be defined.

REQ-APP-02-27. Event type reflecting system shutdown MUST be defined.

5.3 Dependencies Requirements

REQ-APP-03-01. UE descriptor MUST contain elements listing the services which must be available in order for the UE to execute correctly

REQ-APP-03-02. UE descriptor MAY contain elements listing the service versions which must be available in order for the UE to execute correctly

REQ-APP-03-03. UE descriptor MAY contain elements listing the other UE's versions so that these UEs can be launched by the current one

REQ-APP-03-04. UDP descriptor MAY contain elements describing the versions of native libraries it contains

REQ-APP-03-06. If the requirement ***REQ-APP-03-04*** is fulfilled, the MEG framework MUST be able to return the current version installed for some native libraries

REQ-APP-03-07. The UDP and UE descriptors MAY contain elements describing requirements for physical resources

REQ-APP-03-08. If the requirement ***REQ-APP-03-07*** is fulfilled, the MEG framework MUST be able to return description of physical resources of the device

REQ-APP-03-09. The UDP and UE descriptors MAY contain elements describing requirements for logical resources

REQ-APP-03-10. If the requirement ***REQ-APP-03-09*** is fulfilled, the MEG framework MUST be able to return description of logical resources of the device

REQ-APP-03-11. If the requirement ***REQ-APP-03-09*** is fulfilled, the MEG framework MUST be able to determine dependency resolution based upon keyword matching

REQ-APP-03-12. If the requirement ***REQ-APP-03-09*** is fulfilled, the MEG framework MUST be able to determine dependency resolution based upon matching values for the same names

REQ-APP-03-13. If the requirement ***REQ-APP-03-09*** is fulfilled, the MEG framework MUST be able to determine dependency resolution based upon matching numeric values with specified intervals

REQ-APP-03-14. MEG framework MUST ensure that only a minimum set of UDPs (and therefore classloaders) are active at any given point of time. The set of active UDPs in an ideal state should be equal to the closure of all the package and service dependencies required by the UEs to conserve the available volatile memory in the device.

REQ-APP-03-15A. MEG framework MUST support UDPs that are active at the MEG framework startup and are available to all UEs throughout the execution lifetime of the MEG framework.

REQ-APP-03-15B. MEG framework MUST support UDPs that are lazily activated as services are requested by the UEs during its execution.

REQ-APP-03-16. MEG framework MUST support UDPs that contain zero or more UEs.

Note: There could be UDPs that contains only services and class libraries and no UEs.

5.4 MIDP Requirements

REQ-APP-04-01A. MIDlet execution environment implementation MUST pass the MIDP2.0 TCKs.

REQ-APP-04-01B. MIDlets written for CLDC1.0 and CLDC1.1 configurations MUST run unaltered in the MIDlet execution environment.

REQ-APP-04-02. MIDlets written to MIDP1.0 and MIDP2.0 MUST run without any modification on the OSGi framework. The environment supporting the execution of MIDlet should ensure binary compatibility (recompilation of MIDlet sources SHOULD NOT be required). This level of compatibility is necessary to pass MIDP1.0 and MIDP2.0 TCKs.

REQ-APP-04-03A. MIDlet execution environment MUST allow MIDlets to access selective set of exported classes by the OSGi framework.

REQ-APP-04-03B. MIDlet execution environment MAY allow MIDlets to access selective set of exported services by the OSGi framework.

REQ-APP-04-04. Support for concurrent MIDlet suites MUST be optional. MEG MUST NOT prevent implementing support for multiple isolated simultaneously executing MIDlet suites.

REQ-APP-04-05. MEG framework MUST support MIDP OTA user initiated provisioning as specified in MIDP 2.0 specification.

REQ-APP-04-06. It MUST be possible to include any extension APIs into the platform (e.g. WMA 1.1). As an example, the JCP defined extension APIs within the MIDlet execution environment implementation MUST pass all TCKs defined for it.

Note: This requirement has to do with MIDlet execution environment within MEG framework supporting pluggability of additional extension APIs, rather than specifying the requirement of extension APIs. The requirement on TCK is more for ensuring compatibility of the extension APIs running within MEG framework. Once the RFC begins and high level architecture is defined, perhaps a compatibility & interoperability document detailing these requirements should be defined.

REQ-APP-04-08. It SHOULD be possible to augment the MIDlet execution environment dynamically by installing/updating/removing UDPs containing extension class libraries (either standard JSRs or Licensee Open Class Libraries). By dynamic we mean that as new APIs become available, it should be possible to augment the MIDP execution environment with the new APIs and new MIDlet should be able to take advantage of this API. There is no requirement for an executing MIDlet to become aware of the changes to the class libraries or for new class libraries to take effect during the MIDlet execution.

REQ-APP-04-09. MIDlet execution environment MUST map MIDP 2.0 security model and MIDP 2.0 recommended security practice to the OSGi/J2SE style permissions.

REQ-APP-04-10. From the end user experience perspective, MEG management framework MUST not distinguish between MIDlet and other UEs (it would be confusing to the user otherwise). This requirement does not have anything to do with the behavior of the executing MIDlet. It just implies that from the end user these are essentially similar and therefore treated similarly by the MEG management framework.

REQ-APP-04-11. MEG management framework SHOULD be able to terminate an executing MIDlet.

Reason why this requirement is a not a MUST is because, it may not always be possible to terminate a MIDlet execution (Java's Thread.stop() issues).

All Page Within This Box

5.5 Application Container Requirements

REQ-APP-05-1. MEG framework MUST support pluggable application container to support non OSGi compliant application types, like MIDlet, DoJa, Xlet etc.

REQ-APP-05-2. MEG framework MUST support dynamic add/update/removal of new containers to support additional application types.

REQ-APP-05-3. An application container MAY support more than one application type.

REQ-APP-05-4. Application container MUST ensure binary compatibility for the nested applications so that they can be executed without recompilation.

REQ-APP-05-5. Application container MUST map the differences in the lifecycle between the OSGi/MEG framework and the application running within the container.

REQ-APP-05-6. MEG framework MUST be capable of starting/terminating an application running within the OSGi/MEG framework.

REQ-APP-05-7. Application container MUST map the nested application's security model and security practice to the OSGi/J2SE security model & policies.

REQ-APP-05-8. Applications running within the Application container MAY be allowed access to the OSGi/MEG Services.

REQ-APP-05-9. Applications running within the Application container MAY be allowed to export and import packages.

REQ-APP-05-10. It MUST be possible to install any extension APIs and make it available to the nested applications within the Application container.

REQ-APP-05-11. From the end user perspective MEG framework MUST not distinguish between applications running within the Application Container and MEG specific UEs.

REQ-APP-05-12. MEG framework MUST not prevent concurrent execution of UEs and applications running within the Application Containers.

6 Document Support

6.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

6.2 Author's Address

Name	Harry Prabandham (harryp@motorola.com) and Vadim Draluk (vdraluk@motorola.com)
Company	Motorola
Address	
Voice	
e-mail	

6.3 Acronyms and Abbreviations

6.4 End of Document

All Page Within This Box