Mobile Agent for Nomadic Devices

Charif Mahmoudi, Fabrice Mourlin and Guy-Lahlou Djiken
Laboratoire d'Algorithmique, Complexité et Logique
University Paris-Est
Creteil, France
{charif.mahmoudi, fabrice.mourlin, guy-lahlou.djiken}@u-pec.fr

Abstract— Today, mobile devices are used as personal objects, which contain own data about our activities, our personal life, etc. Also, these data are essential for the user and it is not acceptable to exchange these data on the network. Then, computing on these data can be done by importing applications. We present our work about moving applications from server to client host. We use mobile agent technology with the difficulties of heterogeneous platforms. Importing agent has to respect features of the device (version, technical interface, business interface, etc.). A first negotiation is settled to ensure that a set of useful agents are selected towards an embedded device. By the end of a scenario, features are recorded to improve negotiation algorithm for the future exchange. So, each exchanged agent is not only used for a mission but also for the next ones. We use our framework for collecting data from nomadic devices. The mobile agents bring back these data to server where they will be treated.

Keywords-mobile OSGi; agent; nomadic device; REST services; transcoding.

I. INTRODUCTION

Mobility is a common word, which has different meaning depending on the working context. Mobile feature is often used for device like smart phone, tablet. In that case, mobile aspect highlights that user and device have the ability to move during runtime application. Distributed systems use mobility for actions. Mobile agents are often used for adapting an application to its environment. In that case, it means that the code of a part of an application can move from one node of the network to another one [1].

Two meanings of one word for two distinct contexts seem ambiguous and we should speak about nomadic user but it is too late for changing a so common feature of phone for instance. In our working context both aspects are used to build a distributed application where mobile devices are http client of distributed application.

Development of mobile applications is an activity domain which focuses a lot of designers and programmers. The number of libraries encourages new kinds of interconnection. Several protocols are used depending on the scope of exchanges. Bluetooth [2] and IrdA [3] are exploited for local exchanges, like file transfers with a laptop. A WIFI scanner allows users to connect all WIFI networks. This increases the scope of applications. For example, a client can register a broadcast receiver to perform the scan for new networks and improve its knowledge about environment [4]. Another application context is the developments of Binder framework for inter process communication [5]. It is designed to provide a rich high-level abstraction on top of traditional operating system services

Traditional software architectures use network as input or output data flows. Routes are defined between clients and servers and these are used to exchange information. These routes can be configured dynamically. But when a client is mobile, it enters into an infinite process to adapt current configuration. Also, this update costs time and energy. Another approach could be to update configuration only before communication. But, these events are not always predictable and some of the updates might be useless.

We describe our approach in the next sections of this document. First, we explain in detail our software architecture. Then, we focus to agent server and how agents are exposed over http protocol. Next section is about agent host and the lifecycle of the incoming piece of code. Finally, we describe one of our case studies based on a data collection which gives information to the server about end user activity.

II. RELATED WORK

The Android platform covers a large software solution for mobile devices from an operating system to a set of mobile applications [6]. The two competing: Windows Mobile and Apple's iPhone allows simplified development environment built on proprietary operating systems that restrict interactions between applications and native data. Android offers an open source development environment built on a Linux kernel [7]. Android offers also a standard API to access to the device hardware; this API allows the application to interact Wi-Fi, Bluetooth, and other hardware components.

The Open Gateway Services interfaces (OSGi) platform [8] defines a standardized, components-oriented computing approach for services. This approach is the foundation of a Service-Oriented Architecture (SOA) based system. The OSGi specifications were created to cover the exposition of residential internet gateways for home automation software [9, 10]. However, the extensible features of OSGi technology contribute to impose it as key solutions. Several devices vendors, like Nokia and Motorola, have choice OSGi standard as a base framework for their smart phones. This

 ISBN: 978-1-61208-313-1

choice is due to the features provided secure support for developing and deploying java service component applications packaged in a standard Java Archive (JAR) file called bundles. A bundle contains a manifest file that describe bundle's configuration on the OSGi container. Figure 1 illustrates the OSGi framework, applied to our approach, which provides a shared runtime environment capable of dynamic and hot lifecycle management operations: installs, updates, and uninstalls bundles. No need to restarting the system after a management operation.

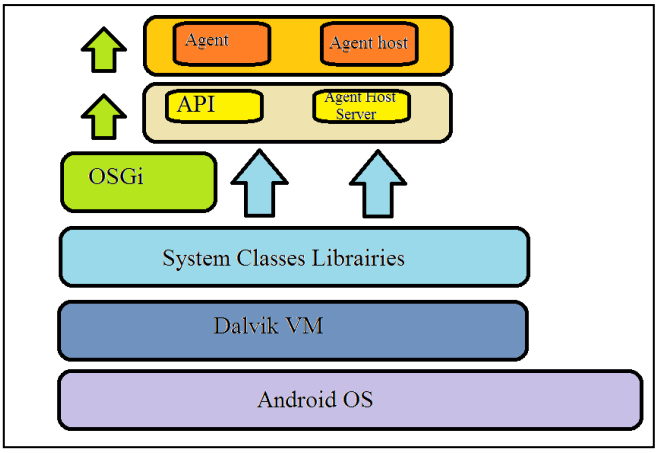

Fig. 1. Agent runtime environment architecture.

Other research works in literature have been focused in the use of agent-based technology on nomadic devices; examples include JADE [11], JaCa-Android [12], AgentFactory [13], and 3APL [14]. Differently from these related works, our approach do not consider the issue of porting agent technologies on limited capability devices, but we focus on the autonomous agent mobility issues, we focus also on the advantages brought by the adoption of an agent-based solution for the development of mobile agents that have decide when and where to move in a complex mobile system.

We propose a new approach based on mobile agents [15] that run into an Android embedded OSGi runtime environment. Our work changes software architecture into a new scheme. When an agent incomes onto a device; it has knowledge about network configuration. This agent collects business data and by the end of its mission, it can return to a server or other clients.

III. MOBILE SOFTWARE ARCHITECTURE

The software architecture of distributed application is a picture of the system that aids in the understanding of how the system will behave. Such an architecture is depicted by a component graph where each node is a software component. A first observation enhances three main types of components: agent server, agent host, and mobile agent.

The role of agent server is to receive the requests from clients then it records the demands and creates or selects mobile agents. Finally, it exports agent to agent hosts. The role of agent host is to send its demand to a server and then listen to the answer of the server. When the reception occurs, it engages the mobile agent into a state where it is able to execute its own mission. To sum up, a mobile agent is a piece of code which travels over the network. Initially, it is configured by the server for navigating through a set of agent hosts. Its aim is to be as autonomous as possible even if it has to use local resources of host client. Security concerns have to be set first on all the devices which will participate to a case study. A more common definition of mobile agent can be found in Bernichi [16].

A more precise observation stresses technical requirements about network exchange and also message structure between the components. Thereby, a client communicates though the use of REST Web Services [17]. This involves that the underlying protocol is http. However, type of message is considered as a byte stream from the hosts to the mobile agent and vice versa.

A. A first level observation

First of all, we provide a deployment diagram where main nodes of architecture are drawn. All connection mobile software support http as the transport protocol, making each compatible for use with the server through a solution on Android devices. No security constraints are applied in the first version of the prototype, but our security approach is developed in subsection III.C.

Server is accessible through WIFI card and it may connect to any standard WIFI router, which is configured as an Access Point (AP), and then, sends the data to other devices in the same network such as basic phones and smart phones. Figure 2 represents the main items: a server which supports an agent server; mobile devices support mobile nodes and agent hosts. WIFI access is provided by antenna or access point. The structure of the graph can be permanently evolved. Of course, other items can be added for example Bluetooth devices.

Once associated with the Access Point, the nodes may ask for an IP address by using the DHCP protocol or use a preconfigured static IP. The Access Point connection can be encrypted, in this case, we have to specify also the pass-phrase or key to the WIFI module:

Nodes may also connect to a standard WIFI router with DSL or cable connectivity and send the information to a web server located on the Internet. Then, users are able to get this information from the Cloud when a static configuration is used.

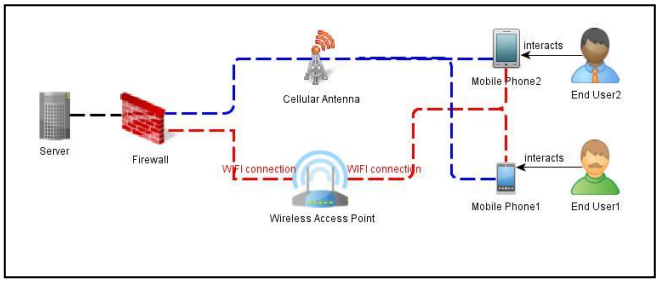

Fig. 2. Deployment diagram of our following case study.

B. A second level observation

After a first physical description, we provide a more synthetic architecture description where components are

isolated. In this context, a component is a modular unit that is replaceable within its environment. Its internals are hidden but it has well defined provided interfaces. Three main components are designed with their dependencies.

The main component is called Agent Server on the server station. It provides two interfaces. One is used for registering the demands of the hosts. Another interface allows mobile agents to exchange data with Agent Server. This interface is particularly important for the end of mobile agent activity.

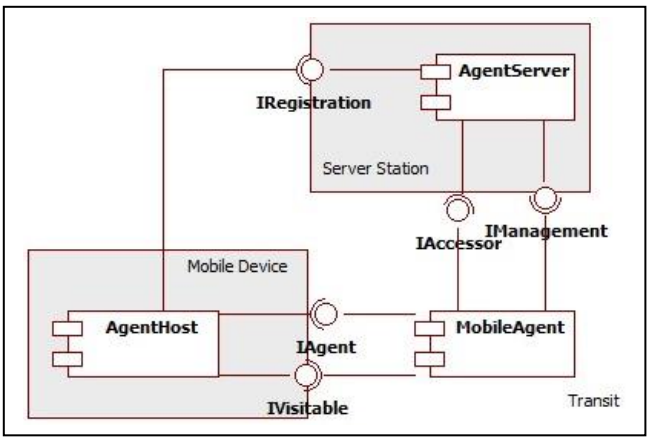

Fig. 3. Component diagram :software architecture.

As illustrated in Figure 3, on each mobile device, a component called Agent Host is installed. It sends demands on mobile services by the use of *IRegistration* interface. This component implements an interface called *IVisitable*. It allows mobile agent to come into the context of agent host.

The component called Mobile Agent, is created and configured of the server station, then it uses *IVisitable* interface of agent hosts for the exportation. It provides an interface called *IAgent*, which is used by the host to launch the runtime of mobile agent on the nomadic device.

The interface *IRegistration* and *IVisitable* are remote but *IAgent* is a local interface. All remote accesses are mapped on http protocol methods. The technology REpresentation State Transfer (REST) [18] is chosen. A resource-oriented approach is well adapted to the interaction scheduling. A resource is a mobile agent for the agent server or an agent host. Or a resource can be an agent host for a mobile agent. It means anything of potential interest that is serializable in some form. The acronym REST refers to the transfer of some bit of information or mobile agent state, as a representation, from a server to a host or back again.

But, another technical constraint appears: the kind of byte code is not the same between the server station and the mobile device. Also, an agent which is built by the agent server on the server has to be transcoded before moving to an agent host. The transcoding means to know the technical features of all agent hosts. Also, we decided to encode mobile agent into an intermediate representation and each local host will adapt the representation though the use of an agent loader. This strategy is close to class loading but the algorithm is not only about loading the byte code of the agent but also on the permissions that are assigned to the agent.

C. security approach

When agent have to access to a resource exposed by current agent host, the agent use URI to address and get the resource. Android SDK 1.0 introduced a security mechanism to manage URI based security. An agent can specify resource URI identifying an XML file. If the agent doesn't have read permission to the agent host containing the XML file, the agent can use its own URI permission instead. In this case, the agent uses a read flag that grants the agent host access to the resource. URI permissions are essentially capabilities for agent execution. This mechanism allows a least privilege access to the agent host. The tractability of policy is preserved truth the agent host.

In the next section, each component is described to illustrate the mechanisms of code mobility and monitoring the collection of information. The structure of the API allows readers to understand the case study presented is the following example.

IV. AGENT EXPOSITION

The role of each component is crucial in defining this new software architecture. Also in this section, we will focus on the structure of each and how is implemented software mobility.

A. Agent Server component

The *AgentServer* component manages the demands of agent hosts and the pool of mobile agents. Its first role is to receive queries from agent hosts and treat them. To do this, it is necessary to detail the structure of a query. It provides two main records: the need for intervention, a reference for the visit. The intervention of a mobile agent means a remote activity is requested by an agent host. This expression is made by the demand of a technical interface, which is associated to permissions for access to local resources. As part of this work, we have not installed directory services where these interfaces have been reported. This should be done in a real environment.

1) Design constraints

We show in section C the structure of a mobile agent. We have decided that all mobile agents can provide only one business interface for the purpose of simplification and optimization. In fact, offering one business interface allows the server to configure a mobile agent that travels to several agent hosts. Each agent host has provided its reference in its request. Then, a mobile agent manages a list of references to agent hosts.

An agent server not only has the role to create mobile agents as well as receiving mobile agents by the end of their mission. This means having traveled all agent hosts; a mobile agent ends at the agent server. Both activities are dependent and yet receiving mobile agents must take precedence over creation because the pool of agents waiting mission is to be reused for future requests. To implement this constraint, we have defined an agent server as composed of two main threads. The first is to the end of mission, the second for the creation of agents. This thread is of lower priority to the first interrupt by creating a return.

 ISBN: 978-1-61208-313-1

When returning mobile agents to the server, it is responsible for the extraction of data and their backup in the local system of persistence. In our case study, it is a database server. These data are then used by other applications so that future interventions will be planned. For example, when collecting data for monitoring Web server, the messages on anomaly deployment involve the creation of new demands of mobile agent. So, a mobile agent will be exported onto the host where the deployment fails. This update will involve new messages during the next data collection and so on.

In our prototype, all agent hosts are treated equally by the agent server. However, the result of the activity of mobile agent cannot be validated by agent host itself. Also, this could be the role of other mobile agents into a control activity. The goal at the server level is to ensure that any received request by the server is handled by a mobile agent. In another work context, it would be possible to keep track of a set of requests to consider exporting mobile groups of agents in a single export.

We did not secure the exchange of data between the server and mobile agents in the context of this work. We considered that the data collected by mobile agents were visible to all components of the distributed system. This simplification allows us to reach a rapid prototype supporting our functional tests, but also our load tests.

2) Component architecture

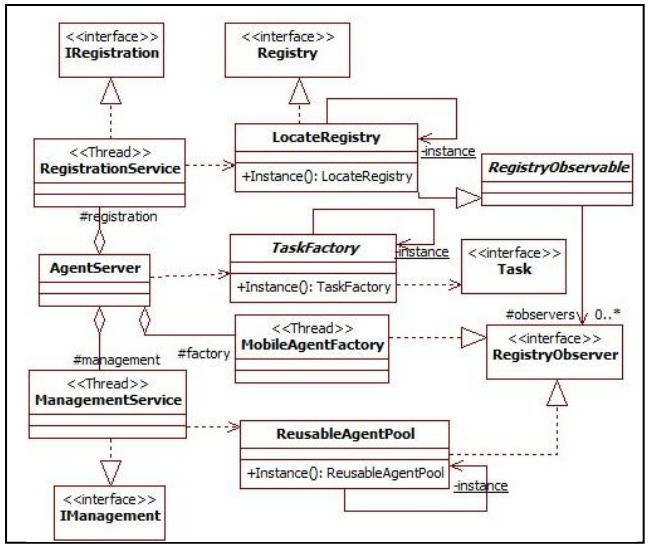

Fig. 4. Class diagram of agent server component

Figure 4 gives a first structure of the classes which belong to the *AgentServer* component. New interfaces are present such as *Registry*, *RegistryObserver*. These interfaces are not exposed by the *AgentServer* component, but are useful as local interfaces within the component itself. Thus, *RegistryObserver* interface is implemented by all classes that are prone to react to requests from a host. Implementation classes are the pool of mobile agents and the class of mobile agent factory.

The class diagram above highlights some design patterns. The data structure that records the host requests is an observable subject by all observers who treat them. The factory of mobile agents is an observer but also the pool of mobile agents ready to be configured for a new mission. The uniqueness of some essential items in the *AgentServer* component is implemented by the use of Singleton pattern. This is the case for *LocateRegistry* class has only one instance for recording all requests from the host.

Technical classes are not present on the diagram in Figure 4 for clarity. For example, management of tasks requested by the host is missing from the figure. In addition, the agent server uses a task set to inject during the configuration of mobile agents. This means the injection of code into an agent, which is already created, but also initiates an initial context for the task performed by the agent.

B. Agent Host component

By definition, the host agent is installed on the nomadic device such as tablets and smartphones. In our current prototype, we have chosen to implement an agent host per device. Thus, the host can be seen in the distribution as representative of the software device that supports system. This component manages the identifiers among other material information.

1) Analysis requirements

A host of agents exhibits a single remote interface to be visited by requested mobile agents. The advantage of this approach is better interactivity. Thus, the interface is published in the directory of the host interface (*Registry*) only when the request from the host has been accepted by the server. In addition, this interface will be removed as soon as the mobile agent has visited the host.

The expression of the need for intervention is difficult because it is necessary that the host is aware of the types of jobs available on the server. As part of this prototype, we made the choice to have a set of interfaces known agent hosts and agent server. It is clear that this global knowledge is not desirable because it all tipsy scalability tasks. More particularly, this approach prohibits the dynamic task creation. This concept will be addressed in the next increment of our prototype.

The structure of an agent host has the particularity to execute a business process but also to involve a mobile agent. This would happen following a request from the host. It is important at this stage to focus on the principle of mobility that we have implemented. A strong constraint is the byte code differences between the platform server and nomadic devices. A host agent cannot execute code from a mobile agent created on the server. It is necessary to transcode this code to adapt to the device from which the request comes. The technical risk was initially important in our project and encouraged us to assign our increment to it. This conversion is done on the server. When the agent host receives the byte code of the class of the mobile agent, it must load it via a local agent loader and configure from the state of the serialized (with its class) agent.

Then, the mobile agent is executed in accordance with the life cycle of the host agents. This means that a thread is dedicated to the mobile agent and activity. The priority of this thread is normal to let the opportunity to agent host to launch higher-priority threads. Finally, specific permissions

 ISBN: 978-1-61208-313-1

are assigned to *AgentHost* component in order to perform network communications and access to some useful sensors for the business job.

2) Design patterns

The main structure of an agent host is based on the *Command* design pattern. Figure 5 shows that design choice. When a mobile agent invokes *IVisitable* interface to enter the host, the client of this pattern is the *VisitableService* service. It creates a concrete *MobileAgent* instance from input byte data and sets its receiver.

The class called, *AgentLoader* promotes the byte code array into a class which can be used by agent host. The agent thus received is added to the instance of *WaitingRoom* class is the data structure that are placed all mobile agents received awaiting use. This instance acts as an *Invoker*. It asks the concrete mobile agent to carry out the activity. *MobileAgent* abstract class declares an interface for executing the task. *CollectorAgent* and *MonitorAgent* define a binding between a *Host* object and a task. It implements methods of public interface *IAgent*. We use the concrete command *CollectorAgent* into the next case study. Its role is to invoke the corresponding operation on host. In the next example (section V), the data are extracted from data files. The *Host* class knows how to perform the operations associated with carrying out a request. This class serves as a receiver for all the concrete commands.

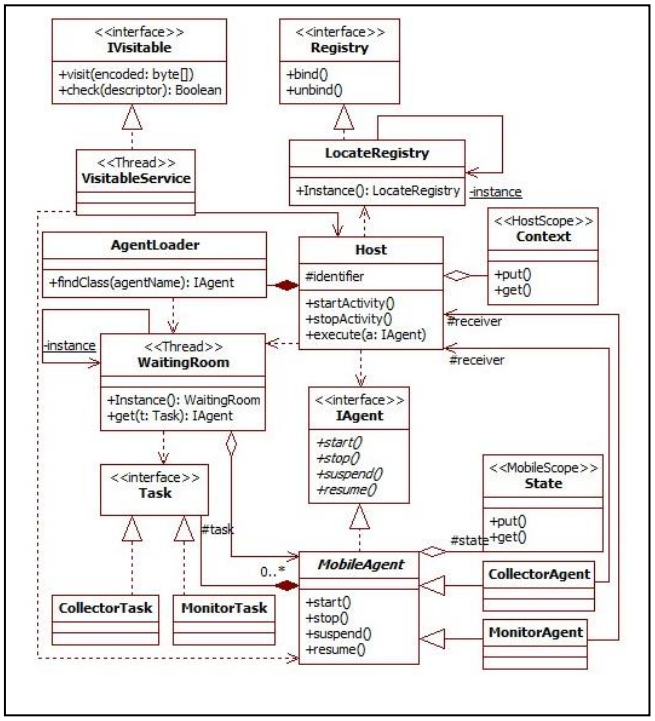

Fig. 5. Class diagram of agent host component

A *Host* object represents a mobile location (any possible place) where a mobile agent may run. Thus it is represented as a physical resource with computing capabilities. An instance of this class is always bound with the corresponding native host. Several local information are stored by the host in order to be recognized as unique in the wireless network without using information specific to the protocol itself.

The *Context* class (Figure 5) shows the data structure used by mobile agents as they pass on the host. Two scopes of data are useful. First data *host* range: they are resident data that the mobile agent can take starting from the agent host. These data are useful for its current activity and the realization of its mission on this host. *Mobile* scope data are those that can move with the mobile agent during its travel from host to host. Usually, these data are the overall mission must make the mobile agent. By the end of the mission, the extraction results come from scope *mobile* data. They allow an external application to validate the success or not of the mission of the mobile agent.

C. Mobile Agent component

The concept of mobile code we use; is a mobility on demand, because it is controlled by the applicants. In our case, the server and the hosts are the assets of our distributed system. Hosts request and the server provides services. But in our case the product is not a set of raw data, it is a code that intelligence can make a mobile service. By configuring the server, the agent knows the references hosts and thus moves from host to host in order to apply the only task that knows. Of course, this task can use specific *host* data or specific *mobile* data.

1) Mobility on demand.

This expression is often associated to electric vehicle in a city. But, in our working context, this represents use of mobile agent in a distributed system. To be useful the mobile agent needs the host offers an agent loader. Then, its statement is managed by the lifecycle of the host. Thus, it is the host that will decide the launch of its task or its interruption, etc.

The end of normal task of an agent is reached the *stop()* method is executed. In order to comply with a protocol of the easiest possible use, we decided to adopt a balanced approach than the arrival of the mobile agent on a host. Thus, the execution of the *stop()* method is followed by the data backup (local to the host or mobile), and then moving the agent to its next destination. The last destination will be the agent server. Once the migration is done, the code of the agent is discharged. And two successive interventions of an agent of the same type will be considered as two different interventions.

When looking for the next host, the mobile agent performs a search of the reference that it has received from the server during configuration step. There can be only one registry per host. In our case, each host has permissions to manage its own registry but can only lookup into abroad registry. Also, the search is performed by a multi cast on all of these registries to find out the reference of the next destination host.

In the context that such a reference is not found then the mobile agent fails to migrate to the next destination. If this reference is last then he will go on the server that originally configured. The final extraction of the data is made by the server. It calls the *IAccessor* interface that is managed by the component *MobileAgent*. We designed this refund so that all the data is aggregated results in a data structure which only read access is possible through the use of *IAccessor*

 ISBN: 978-1-61208-313-1

interface. At its next configuration, the contents of these results will be deleted from the memory of the mobile agent as for all *mobile* data scope.

On the server, the management of mobile agent in done by the data structure called *ReusableAgentPool* which is an iterable structure. We have not developed a technique to eliminate redundant in the waiting pool agents. Another strategy recommended in the work of load balancing is to let the agent on the last host who asked. The underlying idea concerns the fact that an agent is more useful from a client to a server.

2) Design and change.

The mobile agent design is simple as it is important that the agent is small so it can be easily encoded and decoded. By construction, a mobile agent is a running activity in the context of remote work. All types of work are not present on the charts but simply *Task* interface.

To be independent, a mobile agent must be configured. It comes from the server contains a list of steps that are the mission of this agent. A step is the application of a task on a host. By the end of the application of a task is achieved and added to the result list (class called *ResultManagement*).

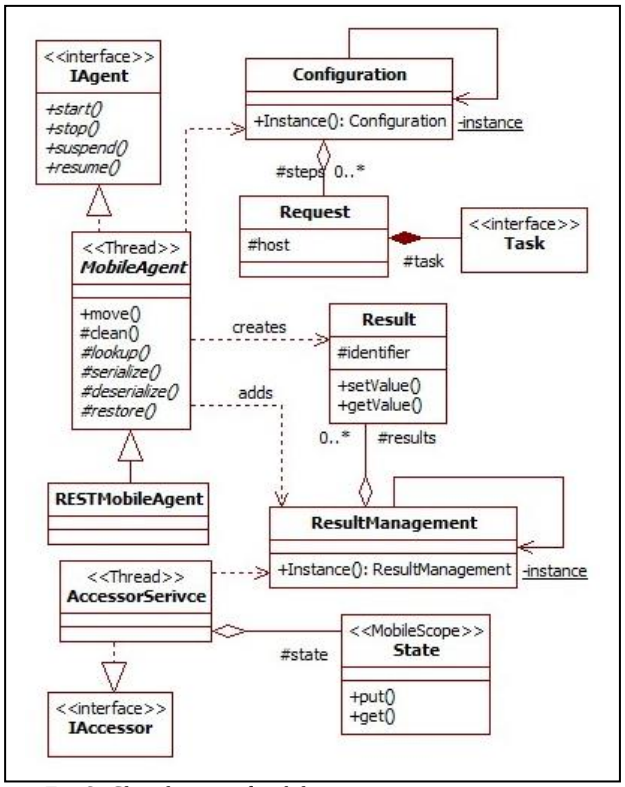

Fig. 6. Class diagram of mobile agent component

Thus, by the end of the mission, the list of steps is empty while the list of results is full. Each of the support structures of an iterator that enables an enumeration of the data structure in the same order as that of the host list.

The strategy of migration is achieved by the *move()* method. This is a step algorithm which is realized by the use of *Template* design pattern. These steps are implemented using abstract methods. Subclasses change the abstract methods to implement the real actions. Thus the general algorithm is saved in the class called *MobileAgent* but the concrete steps may be changed by the subclasses. The refined implementation is done in *RESTMobileAgent* class, where all the steps are implemented as REST web services (Figure 6).

The REST approach is oriented around resources. The resources support often access though get, post, put or delete actions. We built a whole REST prototype, but our design could support other kinds of remote access if necessary.

Next section is about how mobile agent activities are inserted into the life cycle of agent hosts.

V. AGENT HOST LIFECYCLE

Deploying the set of components, they are started at installation. And any agent host is ready to receive a mobile agent after an initial setup phase. Management of mobile agents or constraints specific to each host and belong to the configuration of the host. This is done through the use of XML descriptor.

A. Component descriptor

The Mobile Component Descriptor (MCD) describes the properties of agent host. The MCD contains following required information such that the technical name of the component, the version and a description of the component. Specific features are added such that component type (Server, Host, Mobile), runtime environment, etc. Next, the behavior of the component is described as phases of an automaton. The way to specify them is extremely simple and is divided into 4 parts: parameter, transport receiver, transport sender, phase order. These parts are included into `<agentHost/>` tag.

1) Parameter

A parameter is a name-value pair which is used by the component. Each and every top level parameter is transformed into properties in Configuration instance of component. The correct way of defining a parameter is as follows:

```
<parameter name="identifier" value="HostD1"/>
```

2) Transport Receiver

Depending on the underlying transport on which agent hosts are going to run, different transport receivers are defined as follows:

```
<transportReceiver name="http"
class="fr.upec.lacl.device.host.ReceiveController">
    <parameter name="protocol" value="http"/>
    <parameter name="port" value="8888"/>
    <parameter name="version" value="HTTP/1.0"/>
</transportReceiver>
```

The "`name`" attribute of the `<transportReceiver/>` element identifies the type of the transport receiver. It can be HTTP, TCP, etc. But, because we use Android device, HTTPf is selected.

When the host starts up the "class" attribute is for specifying the actual java class that will implement the required interfaces for the transport. Any transport can have zero or more parameters, and any parameters given can be accessed via the corresponding transport receiver.

 ISBN: 978-1-61208-313-1

3) Transport Sender

Like the previous section, Transport senders are registred in the configuration of the host. And later at the runtime, the exportation of mobile agent will follow this feature, we defined HTTP as transport.

```
<transportSender name="http"
 class="fr.upec.lacl.device.host.SendController">
    <parameter name="protocol" value="http"/>
    <parameter name="port" value="8890"/>
    <parameter name="version" value="HTTP/1.0"/>
</transportReceiver>
```

The sender can have zero or more parameters. In the frame above, the port is defined and also the schema of the transport protocol.

We have chosen the same protocol in both cases, but we think about protocol adapter between nomadic devices. For instance, we think about protocol adapter between nomadic devices and the use of Bluetooth protocol as transport protocol and OBEX serialization.

4) Phase Order

Specifying the order of phases of an agent host in the execution chain is essential to know when mobile agent can interrupt the host.

```
<phaseOrder name="lifecycle"
         package="fr.upec.lacl.device.host">
 <phase name="init" type="start" class="Init"/>
 <phase name="business" type="loop">
  <handler name="observer" class="Display"/>
  <step name="step1" type="exec" class="Wave1"/>
  <step name="step2" type="import" class="Import"/>
  <step name="step3" type="exec" class="Wave2"/>
  <step name="step4" type="export" class="Export"/>
 </phase>
 <phase name="end" type="stop" class="Close"/>
</phaseOrder
```

Each phase can have a handler, which observes or displays details about the phase. In the example above, the class *Display* has a method called handle(), which does the observation.

A phase is defined by a name and a type which corresponds to an event in the behavior of the agent host and an implementation class. For instance, the phase named *init* has a type called *start*. It means that the *startActivity()* method of *Host* class (Figure 5), triggers its behavior defined in class called *Init*, and its *doWork()* method. All the phases are defined in the same manner. Thus, the phase named *business* represents the core of the agent host. Its type implies that this is an infinite loop which is subdivided into a sequence of four steps. The first one, called *step1*, is defined by a class called *Wave1*. This is triggered by the *execute()* method of *Host* class. The following step allows host to import a mobile agent. This step is leaded by the *Import* class. The third step corresponds to the end of the business activity of the agent host. This is defined by the *Wave2* class. Finally, the fourth step allows host to export agent if its mission is ended. Otherwise, this step is blocking until the end of the behavior of the mobile agent.

The end of that loop is achieved by the use of interruption from *execute()* method. Then, the phase named *end* is achieved. As before the type called *stop* is bound to *stopActivity()* of *Host* class. So, it triggers the behavior coded into the class called *Close* and its method *doWork()*. This brings a set of technical classes which are not on the figure 5, but they represent a *State* design pattern where each state of the state chart is defined by a class and polymorphic methods. Two steps are dedicated to mobile agents: import and export. Between those events the mobile agent is used by the host.

B. Mobile agent as activity resource

During the execution of a mobile agent, we can consider it as a thread. As all threads on the nomadic device, it has access to resources, which are internal or external, depending on the permissions it possesses. Because it is not possible to add permissions avec the agent is arrived, it is important to prepare its arrival. This step is called negotiation between the mobile agent and the agent host.

From the side of the agent host, it needs to receive an agent able to do a technical interface. On the side of the mobile agent, it needs to have access to resource which belongs to the device. If the mobile agent comes onto the host and discovers that it cannot do its job, time is wasted and computing resource are used for nothing. Also, mobile agent has a description of the resource, it needs to read or write. As an example, we give below the resource description of the collector agent (Figure 5).

```
<resources agent="ca1" class="CollectorAgent"
    package="fr.upec.lacl.device.mobile">
 <resource name="contacts" mode="read"
    uri="content://contacts/people/"/>
</resources>
```

The resource list contains only one resource which is used as a reader. This resource is defined by an URI which is parsed by the host to know whether it is possible to read. When the condition is validated then the migration can happen.

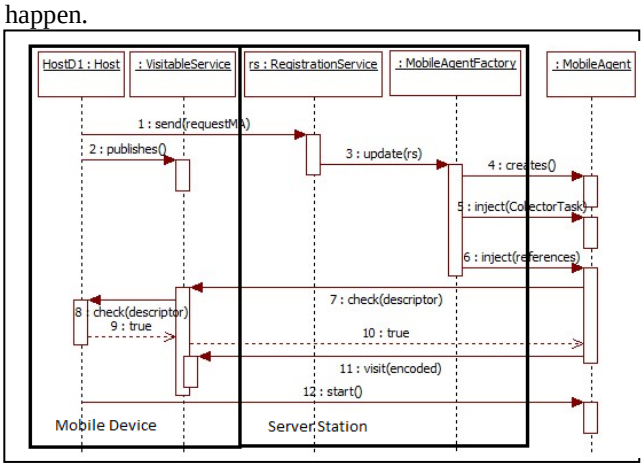

Fig. 7. Interaction diagram of negotiation protocol

To sum up our negotiation protocol, we show the following interaction diagram which is applied at each migration action. After sending its request, the host registers its proxy and waits until a mobile agent is ready (Figure 7).

After configuring a mobile agent, the server injects a task and a list of references into the mobile agent. Then, it can enter into negotiation with the first host of its reference list. The stimulus 7 (Figure 7) tests whether the collector activity is possible onto the host. Because all resources are declared

 ISBN: 978-1-61208-313-1

with uri string, the host parses each of them and decides whether its resources are available.

In Figure 7, the answer of the host is true; also the mobile agent can starts the visit (stimulus 11). At that point the mobile agent is viewed as a binary resource which is read by the read and transformed into a class of agent. This is the role of the *AgentLoader* class which is not design on the Figure 7.

During its runtime, the mobile agent will access to the resources, which are declared into its own descriptor. Because the previous check is satisfied by the host, it means that not only the host provides a way to access to them, but also it ensures that the permissions of the host are sufficient to realize this resource accesses. At that point the roles are opposite, the host can be considered as a resource manager for the mobile agent.

When the task of the agent is finished, it saves its result and looks for the next host reference. If the reference exists, then the stimuli from 7 to 12 will occur again (Figure 7) with this new host, and so on.

C. Instrumentation

In order to identify the root cause of bad behavior, instrumentation has to be introduced. We are interesting into two kinds of anomalies. First kind is about phase tracking. Because hosts are defined as automaton, we want to ensure that the event traces are precisely what is predicted. Second kind is about performance of mobile agents. It is not easy to predict the number of mobile agents is necessary for a set of tasks. If twenty hosts send the same request, only one mobile agent can be sufficient. But, depending on what it has to do, the size of this agent can grow and the serialization and deserialization will cost more time. Moreover, the work of one mobile agent will last more time because, all visit will be done sequentially. Also, it could be interesting to create and configure a set of mobile agent, but how many mobile agents?

These measures need to use external libraries. For the host phase tracking, we use JMX API [19], which is a standard for management and monitoring of resources such as hosts and mobile agents. During the following case study, we observe phases, firing transition, configuration of agents and the notification of state changes during the data collection. We have defined *MXBean* classes which are managed remotely by the *MXBeanServer* of the JVM (Java Virtual Machine) of the server. The tool *jvisualVM* (Sun/Oracle) is used to display their results and allows users to interact with our distributed system.

The time spend within code fragments of mobile agent is interesting to find a limit into the use of mobility. It is important to detect the threshold where one mobile agent costs more time than two. Again, we can observe the impact of the *mobile* data on the migration action. This can involve changes in the management of mobile agents on the server. We use JETM [10] Java Execution Time Measure, which perfectly fits in this kind of time measure. It can be used declaratively and programmatically and collecting data are recorded in a flat or nested manner. An advantage is an http console on port 40000 which is used to visualize execution timings in the form of a dynamic report. We have injected *EtmPoint* instances into the methods of mobile agent and task and we follow all the steps of the behavior of a collector agent.

VI. CASE STUDY

Our work has several technical aspects which are necessary to validate through a case study. This case study has to be understandable even by a non-developer. Also, we have chosen to collect data about the personal contacts recorded into a smartphone or a tablet. The example has several advantages. First, everyone knows the concept of contact into a phone book; this is a tuple of string and number. The size of a set of contacts can be big enough to raise exceptions during the data transfer.

Secondly, this resource is easily used through the use of uri, permissions about it are well known and a whole contact is serialized automatically. Finally, it is easy to check whether our tests are checked, failed or in an error status.

A. Synospsis

An experiment in developing small mobile phone application is not new, but in our context the architecture is more complex. There are a standard server workstation and four nomadic devices. Software is installed on the server to deploy Web services in REST technology. It means Apache Tomcat and Jersey libraries.

First, we have defined a test suite composed on four tests; each of them managed a different strategy of mobility. The first test uses only one mobile agent which travels aver the four devices.

Secondly, we have increased the number of contacts on the devices. Again, this test suite contains four tests where the size of contacts is higher each time.

Finally, we have tested anomaly in case of the descriptor is not compatible with the host. the descriptor is not compatible with the host.

B. Measure and trace event

All the time measures are expressed into millisecond (ms). Because data set are difficult to read, we present only extremes.

1) First test suite about mobility strategy.

a) One mobile agent for 4 agent hosts: this array shows only the bounds; we can note that the serialization costs quite the same time as the task of the mobile agent itself. The same remark is true about deserialization. Also, in that case, it could be interesting to limit the sequence of actions of the mobile agent. A better solution could be to launch in parallel several mobile agents.

b) Two mobile agents per two agent hosts: in that context, the measures are more difficult to exploit because each mobile agent has its own array of result, also it is necessary to aggregate the results and use a global reference to the clock of the agent server. If the whole data collection spends less time than in the first case, the number of serialization are strictly the same but distributed over the 4 agent hosts. We observe that a global time measure from

agent server shows that two mobile agents work faster than only one. But, this gain comes from the distributionof mobile agents. Now, a whole time measure is not a basic sum of all steps. Two partial collects are done in parallel and interesting results come quickly with two mobile agents.

Measurement Point	#	Average	Min	Max	Total
MobileAgent::lookup	4	2,025	1.101	2.910	8.103
MobileAgent::move	4	2.886	2.131	3.982	11.546
MobileAgent::serialize	4	2.992	2.202	4.002	11.970
MobileAgent::deserialize	4	3.045	2.252	4.062	12.180
MobileAgent::start	4	4.757	4.632	4.914	19.028
MobileAgent::stop	4	0.920	0.821	1.001	3.683

Fig. 8. Data results for one mobile agent

2) Second test suite about volume of data set.

Now there are four tests where each agent host has the same number of contacts. But, for the second test, we have doubled the number of contacts per agent host. For the third test, we have multiply by three, and so on. All the data collections are done by two mobile agents as before.

a) Each agent host manages 100 contacts: two mobile agents collects them. This case corresponds to the last experiment (Figure 9).

b) Each agent host manages 400 contacts: the same number of mobile agents collects data. We observe that the duration of the task is quite the same in all the test cases but the serialization and deserialization steps are more expensive. Also in case of the data size is huge, we note that it is essential to increase the number of mobile agents. Thus, the size of mobile data will be bound and the cost of the serialization could be constant.

Measurement Point	#	Average	Min	Max	Total
MobileAgent::lookup	2	1,862	1.113	2.711	3.724
MobileAgent::move	2	2.901	2.811	2.991	5.802
MobileAgent::serialize	2	2.463	2.412	2.515	4.927
MobileAgent::deserialize	2	2.502	2.289	2.715	5.004
MobileAgent::start	2	4.892	4.872	4.913	9.785
MobileAgent::stop	2	0.825	0.823	0.828	1.651

Fig. 9. Data results for two mobile agents

C. Interaction between agents

In the previous, tests we developed cases where there is only one mobile agent per host. Also, no conflict is possible between their activities. But if the data set is too important, we can think about test case where more than one mobile agent will be received by an agent host. So, the first mobile agent could collect a part of the contacts and the second one could collect the other part.

This scenario involves knew ability for agent host and mobile agent. First, if several mobile agents are present on an agent host, each of them should have to manage its own data without any perturbation from the other agents. In that case, the agent host should have to have one agent loader per agent host. So, each mobile agent will be separated by construction. Secondly, if more than one mobile agent realized a task, they have to exchange information or share flags about their own activity. For instance, the first agent collects the first part of the contacts (from one to hundred) and the second collect the next hundred contacts. The cost of the serialization becomes predictable, but mobile agents have to exchange messages during their execution.

This concept of message is implemented in Android framework through the use of Intent service. But, mobile agent comes from an agent server which is not under Android. Also, we have to develop a layer of exchange on the agent hosts to allow mobile agents to have better cohesion.

VII. CONCLUSION

In this paper, we have shown that it was possible to use mobile agents which are interoperable between a JVM and a Dalvik virtual machine. Our work was applied in the context of a data collection. This is a famous example useful in a lot of cases. We have applied an approach of transcoding to adapt byte code from JVM to DVM and vice versa. Measures are computed to highlight that it is essential to configure precisely the pool of agents.

Finally, we have stressed that it was useful to have a message system local to agent host to allow synchronization between mobile agents. The use of a message system global to the device seems to be a solution to explore in future experiments.

REFERENCES

[1] P. Braun, and W. Rossak, “From client-server to mobile agents, mobile agent basic concept, mobility model and the Tracy toolkit” Heidelberg university, Germany, Morgan Kaufmann Publishers, pp. 419–441, 2005.

[2] Bluetooth, S. I. G. Specification of the Bluetooth System, version 1.1, 2001, http://www. bluetooth. com, Retrived October 2013.

[3] S. Williams, “IrDA: past, present and future”, Personal Communications, IEEE, vol. 7, no 1, pp. 11-19, 2000.

[4] S. Li, “Professional Jini: from programmer to programmer”, Wrox Press Publishers, August, 2000, pages.1000.

[5] T. Schreiber. Jacobs and C. P. Bean, “Android Binder, Android inter process communication” Ruhr University, thesis Academic, 2011, pages. 154.

[6] Android Platform Official Site, http://www.android.com, Retrived October 2013 .

[7] J. Chen, P. H. Chen and W. L. LI, “Analysis of Android Kernel,” Modern Computer, Vol. 11, 2009.

[8] OSGi Alliance, OSGi service platform, core specification release 4. Draft, July 2005.

[9] C. Lee, D. Nordstedt and S. Helal, “Enabling smart spaces with OSGi”, IEEE Pervasive Computing 2 (3), pp. 89–94, 2003.

[10] K. Myoung, J. Heo, W.H. Kwon and D.S. Kim, “Design and implementation of home network control protocol on OSGi

 ISBN: 978-1-61208-313-1

for home automation", in: Proceedings of the IEEE International Conference on Advanced Communication Technology, vol. 2, pp. 1163–1168, July 2005.

[11] M. Berger, S. Rusitschka, D. Toropov, M. Watzke, and M. Schlichte. "Porting distributed agent-middleware to small mobile devices." In AAMAS Workshop on Ubiquitous Agents on Embedded, Wearable and Mobile Devices .

[12] S. Andrea, M. Guidi, and A. Ricci. "JaCa-Android: an agent-based platform for building smart mobile applications." Languages, Methodologies, and Development Tools for Multi-Agent Systems. Springer Berlin Heidelberg, pp. 95-114,2011.

[13] C. Muldoon, G. M. P. O'Hare, R. W. Collier, and M. J. O'Grady. "Agent factory micro edition" A framework for ambient applications. In Int. Conference on Computational Science (3) , pp. 727–734, 2006.

[14] F. Koch, J.-J. C. Meyer, F. Dignum, and I. Rahwan. "Programming deliberative agents for mobile services" The 3apl-m platform. In PROMAS , pp 222–235, 2005.

[15] F. Mourlin, C. Dumont, "Implementation of a fault-tolerant system for solving cases of numerical computation", ICIBET 2013, International Conference on Information, Business and Education Technology, ISBN: 978-90-78677-56-7.

[16] M. Bernichi, F. Mourlin, "Two level specification for monitoring application", The Fifth International Conference on Systems, Proceedings of ICONS 2010 - Menuires, The Three Valleys, French Alps, France.

[17] L. Richardson, "RESTful Web Services," O'Reilly Media Publishers, Book pages 220, May 2007.

[18] R. Fielding, "Representational state transfer" Architectural Styles and the Design of Netowork-based Software Architecture, pp. 76-85, 2000

[19] B. G.Sullins, and M. B. Whipple, "Manning JMX in action," Manning Publications Co. pages 424 April 2002.

[20] J. Jenkov, "Java Exception Handling," ProWebSoftware Publisher pages 288, March 2001.

 ISBN: 978-1-61208-313-1