ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

THESIS TITLE

MOBILE TUCSON: THEORETICAL AND TECHNOLOGICAL REQUIREMENTS FOR TUCSON'S PORTING OVER ANDROID MOBILE DEVICES

Thesis in

SISTEMI MULTI AGENTE LM

Supervisor
Prof. ANDREA OMICINI

Presented by
ANTONIO PEDONE

Co-Supervisors
Prof. ENRICO DENTI
Dr. ELENA NARDINI

Session I
Academic Year 2010/2011

A Papà, Mamma, Angela, Lucia

Contents

Abstract

Communication and coordination are two key-aspects in open distributed agent system, being both responsible for the system's behaviour integrity. An infrastructure capable to handling these issues, like TuCSoN, should to be able to exploit modern technologies and tools provided by fast software engineering contexts.

Thesis aims to demonstrate TuCSoN infrastructure's abilities to cope new possibilities, hardware and software, offered by mobile technology. The scenarios are going to configure, are related to the distributed nature of multi-agent systems where an agent should be located and runned just on a mobile device.

We deal new mobile technology frontiers concerned with smartphones using Android operating system by Google.

Analysis and deployment of a distributed agent-based system so described go first to impact with quality and quantity considerations about available resources. Engineering issue at the base of our research is to use TuCSoN against to reduced memory and computing capability of a smartphone, without the loss of functionality, efficiency and integrity for the infrastructure.

Thesis work is organized on two fronts simultaneously: the former is the rationalization process of the available hardware and software resources, the latter, totally orthogonal, is the adaptation and optimization process about TuCSoN architecture for an ad-hoc client side release.

Introduction

Multi-agent systems (MAS) provide an appropriate level of abstraction for modelling and engineering modern systems in order to confront and dominate their growing complexity. The concepts of environment, openness, local control and interaction, are the principal features of modern complex software systems.

Computations and locality influence each other since idea of environment is explicit as well as the interactions with it; systems, designed to be always running, can be changeable in size and structure; system's components are autonomous and pro-active within their locality and they interact on the local basis of spatial and temporal knowledge.

On this background, weak agent definition [WJ95] fits and it is characterized by the concepts of autonomy, pro-activity, spatial location, reactivity and sociality.

Facing complexity in MAS modelling and engineering requires a shift from a reductionist vision of MAS to a systemic, holistic vision of MASs, explicitly accounting for social issues [COZm00, FGM03]: we want to look to a MAS through a lens able to move our attention on social intelligence aspects rather than on an individual intelligence coming out from a single agent [ZJW01, PO02].

MAS aims to deal with: distribution problem over the time and space; new nature of components and their interactions; already cited complexity and unpredictability of the environment. These goal leads to the definition of new abstractions, new meta-models, new methodologies and technologies. Among latter, we study TuCSoN (Tuple Centres Over the Network) infrastructure. It was conceived and designed at research laboratory APICe [API11]: the principal goal of this framework is to support agent communication and coordination providing tuple centres. In management of social intelligence, we reco-

gnize in the distribution over infrastructure's nodes the most important feature of TuCSoN: on these nodes take place tuple centres, which are shared and reactive information spaces.

More specifically, tuple centres are programmable tuple spaces, i.e. tuple spaces with a reactive behaviour which can be programmed dynamically by a specific language. We refer to the coordination logic language called ReSpecT (Reaction Specification Tuples): also designed at APICe laboratory, it is used to define and to express tuple centres behaviour and to manage the interactions, and then the working, of software agents. Agents communicate by creating and retrieving associatively tuples whose existence, once created, is independent with respect to agents' one. This make it possible to obtain uncoupling properties, temporal and spatial, which greatly simplify the engineering of the agent interaction space in systems' development [ALab06].

Finally, we aim to stress that MAS are well designed for Internet context and its aspects. Open distributed multi-agent systems have gained sheer interest due to their suitability to the Internet scenario: infact, their best property is to cope well with the unpredictability and the dynamics of the environment. The lack of a global state of the Internet can be addressed by exploiting the agent autonomy and flexibility [COZc00].

This thesis aims to exploit the possibilities offered by TuCSoN infrastructure through a mobile device, i.e. a smartphone, which perfectly represents, even from a purely practical and visual point of view, the idea of spatial distribution, decentralized control, autonomy and flexibility.

The starting point of our research is represented by TuCSoN, while operating system Android will be the target of our work toward an use's extension of the infrastructure.

TuCSoN's porting over a mobile device Android embedded is the endpoint of the thesis: thanks to a careful analysis of the architecture and capabilities of framework and of operating system, we have co-

me to a TuCSoN release for Android able to satisfy agents' needs, without losing efficiency and integrity.

Thesis follows this organization: first part provides an exhaustive presentation and description of needed resources, TuCSoN and Android, by a discussion of their architecture, their operation and their main issues that we care to deal in order to achieve our ultimate goal; second part is composed by two sections, the former concerns with reasons, analysis and design steps which lead to the porting and the latter presents a case study designed to provide a comprehensive example of the work.

PART I

Needed resources

1 TuCSoN infrastructure

1.1 Design and skills

TuCSoN is an infrastructure providing coordination services for agent-based systems which exploit the Internet scenario. These services are embodied in tuple centres, that are coordination abstractions provided to agents by infrastructure in order to enable and govern agent interaction [OZ99].

Tuple centres are characterized by a reactive behaviour: the agents interact with them by inserting, retrieving and reading information in the form of tuple, ordered collections of heterogeneous information chunks. More specifically, tuple centres are programmable tuple spaces, i.e. tuple spaces with a reactive behaviour which can be programmed dynamically (by humans as well as by agents). Agents access tuple centres associatively by writing, reading and consuming tuples via simple communication operations (out, rd, in, idp, rdp). While the tuple spaces behaviour in response to communication events is fixed and pre-defined by the model, the behaviour of a tuple centre can be tailored to the application needs by defining a suitable set of specification tuples, which define how a tuple centre should react to incoming/outgoing communication events [OR04]. The specification tuples are expressed in the ReSpecT language.

The essential components of TuCSoN are:

- a coordination model based on multiple programmable tuple spaces that mediates all communications among active entities;
- a distributed infrastructure, modelling a system, upon which tuple spaces are deployed;

- the integration of mechanisms for the access control within the tuple-based environment and their application to the hierarchical infrastructure.

From topology point of view, tuple centres are hosted in TuCSoN nodes, distributed over the network, defining the TuCSoN coordination space. It is possible to distinguish two kind of TuCSoN nodes: *places* and *gateways*.

The former represents the nodes hosting tuple centres used for specific applications/systems need, from supporting coordination activities to hosting information or simply enabling agent communication: in a mobile agent framework, places are the nodes where mobile agents are meant to execute. The latter provides instead information for a limited set of places since a single and centralised repository is unfeasible in complex and large environments.

Finally, a *domain* is the set of nodes composed by the gateway and the places for which it provides information.

It is worth noting that the concepts of gateway and place do not automatically imply the definition of a unique hierarchical structure: a place can be part of different domains and a gateway can be a place in its turn. This is useful to model complex and dynamic network topologies, as those deriving from virtual organisations [OR04].

This base model has been extended by the idea of ACC[1], to support the definition and development of organizational structures and their conduct rules. An organisation, in terms of its social structures, rules and resources, is mapped onto a domain. The description of the organisation abstractions and concretions is stored and managed dynamically in a specific tuple centre, called $ORG, which is hosted in the gateway node of an organisation. The $ORG tuple centre hosts then (dynamic) information about societies, roles, agents and related

[1] Agent Coordination Context.

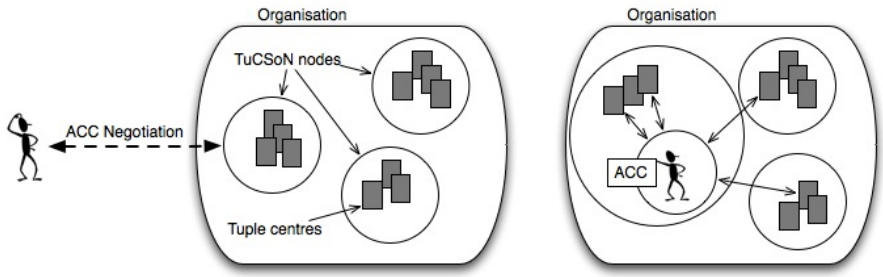

Fig. 1.1. On the left the ACC negotiation, on the right the operating stages for accessing coordination artifacts inside the organisation.

relationships (agent-role[2] e inter-role[3]) defined for the domain represented by the gateway and its places.

Then, in order to access to the tuple centres hosted by a coordination node of a place, an agent must join the organisatoin by entering into a suitable ACC negotiated with the gateway of the domain. Thanks to the basic services provided by the infrastructure, an agent can negotiate the configuration, properties and quality of services characterising the ACC. In the case of successful negotiation, an ACC with a specific configuration is created and entered by the agent, which can then exploit its interface to access and use the tuple centres hosted by the places of the domain [Fig. 1.1]; basically, this interface provides the basic primitives of the ReSpecT coordination language (in, out, inp, rdp, rd, set_spec, get_spec), and enables agent access to tuple centres according to the permissions and rules defined for the roles it is playing.

---

[2] Agent-role relationships: through these relationships it is possible to specify whether a specific agent is allowed (or forbidden) to assume ad the to activate a specific role inside the organisation.

[3] Inter-role relationships: these relationships make ot possible to specify structural dependencias among the roles, so as to further define constraints on dynamic agent-role activation. By means of these relationships, it is possible to explicitly specify, for instance, whether two roles are equivalent, whether a role excludes an othe one, or requires other roles to be played.

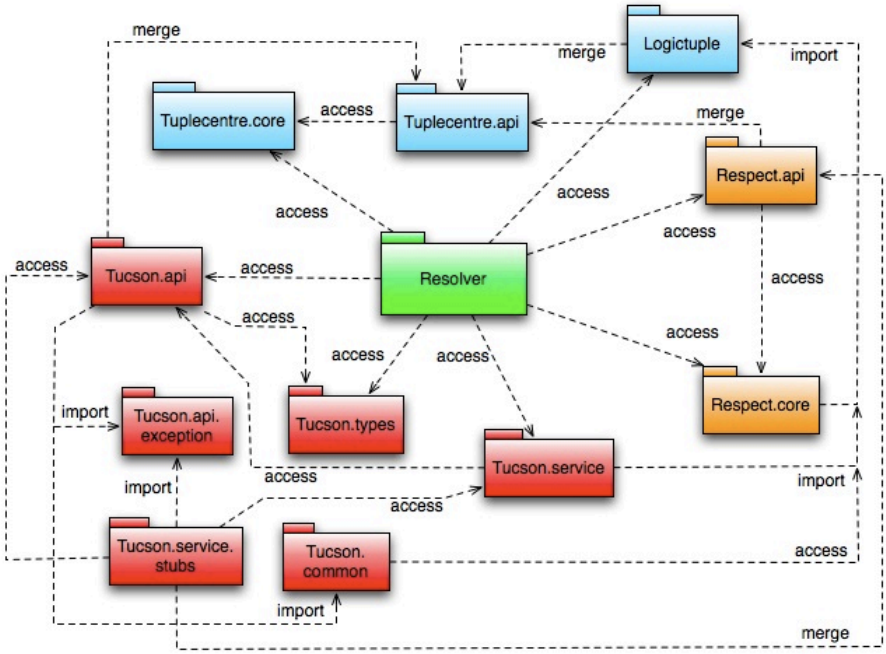

Fig. 1.2. Package diagram of TuCSoN infrastructure 1.9.1.

Also, an ACC provides the primitives to dynamically activate and deactivate roles, and to quit the ACC itself, thus ending the agent working session inside the organisation.

1.2 Separation from ReSpecT

In this section we illustrate the first steps of our research work by focusing on TuCSoN version 1.9.1. Our first studies and attempts of TuCSoN's porting are concerned with the version 1.4.5, while it is the 1.9.1 one which formed the real basis to talking about the separation from ReSpecT and, then, a client side release.

Separation from logic language ReSpecT represents the first piece of the puzzle that will have, as a final result, an Android client release of TuCSoN.

In its previous version (1.4.5) the two technologies, TuCSoN and ReSpecT, formed an *unique ensamble* in which, the former was the management framework for tuple centres and provided adequate support to network distribution, the latter was the coordination language used to characterize a tuple centre.

Version 1.9.1 leads to a redesign of TuCSoN that, on one hand, aims to keep as much as possible the same architectural elements and design properties and, on the other hand, introduces new elements to achieve the prefixes objectives: i.e. the resolution of the separation of languages from the infrastructures issue [SO08].

The parallel phase of ReSpecT re-engineering leads to version 2.2: it's aimed to make such stand-alone technology for the realization of not distribuited concurrent applications[4] and to modify and extend it according to the meta-model A&A[5]. This made it possible to join TuCSoN with this latest ReSpecT version.

Infrastructure package diagram, shown in Fig. 1.2, clarifies how the two technologies are merged by explicating, through dashed arrows, the relationships among the various packages.

UML 2.0 *import* defines a relationship in which a package contains a copy of another package's classifier; *access* refers to a relationship where all source package's classifiers can access to the public classifier of the target package; finally, *merge* specifies a relationship where the contents of the target package are combined with the source ones through specialization and redefinition [NM09].

---

[4] Conceived to program the tuple centres through a specification reaction language (language level), the ReSpecT technology was subsequently equipped with the means useful to build not distributed concurrent applications, in which the tuple centre is the fundamental mean for coordination of different application's tasks (application level).

[5] Agent and Artifact.

Separation of languages from the infrastructures, theoretical reason for TuCSoN version 1.9.1, finds in this thesis a real kind of concreteness.

Indeed, the separation of ReSpecT language from infrastructure, finds now a good motivation to be put into practice. As we will see in second part of thesis, this separation aims to provide two points of view and use, of the infrastructure: on one hand, where a TuCSoN node is installed, we will need all the features and technologies built-in, including ReSpecT, on the other hand, where there will be simply an agent eager to exploit the infrastructure, we'll only need some TuCSoN resources, excluding ReSpecT.

In this context, ReSpecT separation from TuCSoN is seen as structural and conceptual rationalization of the infrastructure, whose beneficiary will be the end user (the agent). An agent, infact, needs only those *APIs* useful to ensure ability to exploit the infrastructure and interact with it: we're talking about those *application programming interface* that make it possible to define and create, specifying an identifier, an agent and a tuple centre. To guarantee those API's use and, at the same time, maintaining the infrastructure's integrity on the node side, we decided to use the classic TuCSoN's API delegating, node side, the bind with the logical language on use. This link between TuCSoN API and ReSpecT lnguage is made by the Resolver in full version 1.9.1.

This separation leads an agent to specify the syntax of name for a tuple centre. Client release will send these information to the node side as a string: the node side will use these received information (along with the canonical other ones provided by the infrastructure's protocol) to create a reference to the tuple centre requested, and then to establish the bind with the respective tuple centre specified by the ReSpecT language.

A final consideration concerns absolute novelty of version 1.9.1, compared to the previous one, to specify a particular language (for the infrastructure and for the agent). By removing computational load of ReSpecT from agent side, we delegate language management to

the TuCSoN node and we released the agent from having to “declare” a logic language. This scenario reverses any responsibility on the node side (except for the syntax check of used names).

Indeed, an agent should be aware only about the valid syntax to specify its own id and the particular tuple centre id with which wants to interact.

The benefits coming from this choice are indispensable for our goal: i.e. to use infrastructure with the least amount of resources and information to possess.

1.3 Toward a client release

After ReSpecT separation step, we focused on the actual possibility of obtaining a light and effective TuCSoN release used only client side, i.e. agent side.

We aim to change some infrastructure aspects in its general form in order to satisfy the requirements coming from our goal. These aspects concerning with initial configuration phase by the tucson_ _conf.properties file, with the language and its management, with Resolver’s operation and, finally, with external libraries use.

First at all, since configuration phase involves a TuCSoN node, we decide to exempt agent from consulting tucson_conf.properties file: infact, it is not an agent prerogative to know the name and locality (i.e. the path) of Resolver, Container and relative language (which are the information contained in the tucson_conf.properties file).

Clearly, these information must be managed, and then learned, by the manager of knowledge which is available to infrastructure.

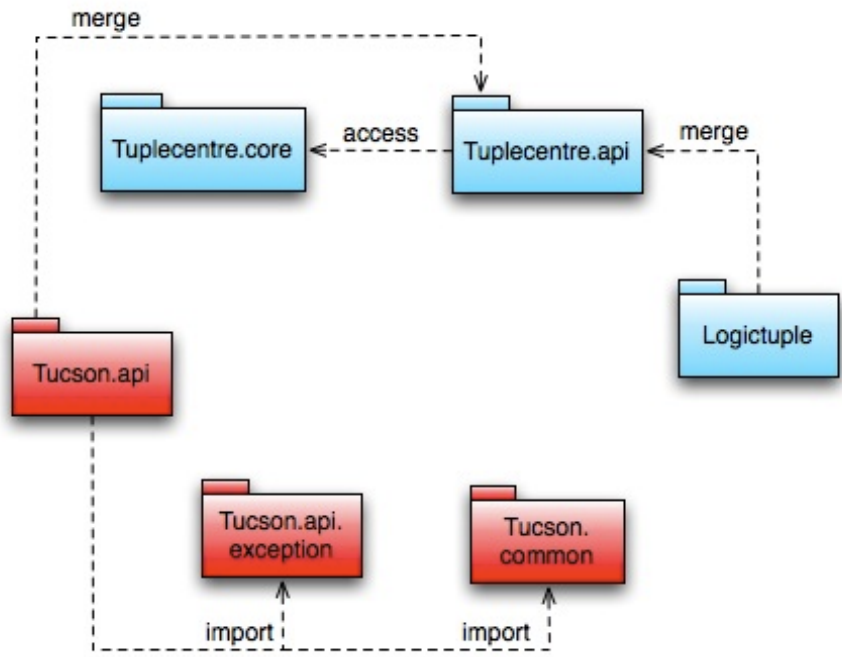

Fig. 1.3. Package diagram of Tucson client release (1.9.2).

All these considerations are strictly related with the available resources: we don't want to install a TuCSoN node on a mobile device, we just want to use infrastructure in a holistic way.

About language management, version 1.9.1 uses a LanguageManager: it is able to verify, with Resolver help, the correspondence between the logic language used by agent and the logic language provided by infrastructure. Once learned infrastructure knowledge (through the tucson_conf.properties file), the LanguageManager is capable to redirect requests coming from an agent on the correct package containing the corresponding language.

Through *reflection* technique, the manager is able to dynamically build links with the methods, useful for agent's needs, that each language should have. In our release, this management is simply bypassed, so agent does not declare any language: all that it has to

know are the TuCsoN APIs and the well-formed syntax for the namespace (for tuple centres and agent id).

On node side, however, infrastructure needs to know which are the available languages. This task is accomplished thanks to already mentioned routine based on Resolver and LanguageManager.

On client side, Resolver loses its skills and its central role due to separation from ReSpect package and from language manager. Resolver is designed to keep any correspondence between a logical language and its users but in this new context it does not find place anymore. Although Resolver is one of the main novelties in version 1.9.1, it is involved in set of routines that should be managed only on node side. Infact, an agent exploiting TuCSoN infrastructure, it has no reason to know which package contains the specific language and how it is called the relative Container.

Fig. 1.3 summarizes these considerations by showing package diagram, much more simpler than the one of whole infrastructure, but also much more functional (in respect to the requirements of a potential client).

About external libraries, our optimization operation concerns *util.jar* and *tuprolog-1.4.jar*[6] [DOR01]. The former is conceived to add frequent and common functionalities, the latter is conceived to ensure accuracy of naming for both AgentId and TupleCentreId and for tuples (structured or not).

New version of library util.jar ensures features designed to extrapolate correctly, for example, the address at which is located a particular tuple centre. Library tuprolog-1.4 (Appendix A) assures to build correctly, in terms of syntax, a tuple or the name of a tuple centre thanks to logical term and logical structure. The OperatorManager also provides syntax checking on the construction of an agent or a tuple centre id.

---

[6] tuProlog is a light-weight Java-based system allowing cnfigurable and scalable Prolog components to be biult and integrated into standard Internet applications according to a multiplicity of different interaction patterns, like JavaBeans, RMI, CORBA and TCP/IP.

As it happens to the following version of tuprolog (Appendix B), util library functionalities can be merged with tuprolog library.

The core of these discussions is represented by need to maintain the correspondence between information sent and received to and from the infrastructure. We try to keep a constant overview of the infrastructure, without ever having to adapt TuCSoN's characteristics to potential agent's needs.

The distinguishing feature of infrastructure is to enable communication and coordination among a very heterogeneous population of agents. Client release, even characterized by an essential simplification and by purchasing a substantial "independence", must be able to preserve this feature. We aim to define a client release able to interface with TuCSoN just like any other node, in order to add dynamism and flexibility, while preserving the general lines of conduct.

1.4 Improving version 1.9.1

Before proceeding with further optimization of client release (already schematized in Fig. 1.3), we summarize all of those studied and re-engineered aspects of TuCSoN version 1.9.1:

- initial configuration phase related with tucson_conf.properties file;
- management of the languages;
- Resolver;
- separation from ReSpecT;
- rationalization of the external libraries.

On the basis of aspects listed above, we further performed an analysis of infrastructure's components (packages): we aim to delete packages which lose any meaning when considered on client side.

First, *tucson.service* package loses meaning because it contains all the structures that offer the possibility to access to infrastructure's services: WelcomeAgent and AgentContextSkeleton are some files of this package which, in a completely symmetric way with respect to

client release, they bring forward the negotiation and the exchange (on node side) of information between TuCSoN and the agent.

Similarly, *tucson.types* package loses meaning because it contains interfaces which declare methods used by Resolver in relation with agent id, tuple centre id and Container; failing Resolver, all these information becomes obviously unnecessary.

Since we have no intention to install a node on agent side, *tucson.test* and *tucson.tools* packages remain outside of client release (not already considered in the previous diagram).

Final rationalization concerns *tuplecentre.core* package: infact, we have eliminated all files related to the operation and administration of the state machine connected to a tuple centre.

2 Mobile world: Android

2.1 O.S. architecture and innovations

Operating system Android represents an important step in *mobile* area of software engineering world. It was conceived to capture the new requirements offered by Internet: i.e. new paradigms for applications development and then new technology and architectural choices.

Information and applications before accessible and executable on any desktop are right now accessible through increasingly powerful devices characterized by mobility and reduced dimensions. Actually so called *mobile device* are really becoming a kind of notebook in which phone function is just one of the many available [Ca10].

Unlike many other operating systems (o.s. henceforth) for mobile terminals developed by main competitors, the most relevant feature that distinguishes Google o.s. is being *open*. First of all, this means that it uses open technologies (like Linux kernel), that libraries and API used for its implementation are the same that we can use to create and/or to extend applications and, finally, that its code is open source: i.e. explorable by anyone wants to improve it, document it or simply understand how does it work.

The license chosen by Open HandsetAlliance [OHA11] is Open Source Apache License 2.0 which allows to different producers to avoid paying any royalties for Android adoption on their own devices.

Although the language used by Android is Java, produced and released by Sun Microsystems, Google has introduced an *ad-hoc* solution to not contradict the open nature of o.s. Infact, we know that de-

vices adopting Sun Virtual Machine (VM), related to J2ME[1] environment, have to pay a royalty: this perspective would be strongly in contrast with Apache license referred above.

Big news of o.s. Android is the capability to use Java language without executing Java bytecode and then, without using a Java Virtual Machine (JVM). Google has adopted an its own VM called Dalvik (Iceland locality's name) in order to optimize as good as possible the limited resources of mobile devices. VM executes code contained in files *.dex* (Dalvik EXecutable) [RiP10] which are obtained, in building phase, since files *.class* of Java bytecode.

Android architecture includes all the stack tools to building mobile applications including an o.s, a set of native libraries for platform's core functionalities, an VM's implementation and a set of Java libraries [Ca10]. From a structural point of view, architecture is organized in levels where the lower ones offer services to the upper ones by providing a higher abstraction's level.

Lowest level of architecture is version 2.6 of Linux kernel: this choice guarantees the presence of low-level tools for virtualization of background hardware through the definition of different drivers. In particular, as can be seen in Fig. 2.1, there are the drivers for the management of multimedia devices, display, Wi-Fi and power. Let's explore in detail the main Android architecture's components.

Above Linux 2.6 kernel layer we find a layer containing a set of native libraries implemented in C and C++ which represents the real Android core. These libraries refer to a set of open source projects.

Surface Manager (SM) is a key component because it manages the views composing the graphical interface. Infact, SM's task is to coordinate the various windows which applications need to display on the screen. Each application is running in a different process and is designing its own interface at different times.

---

[1] Java Micro Edition is a Java platform designed for embedded systems (mobile devices are one kind of such systems).

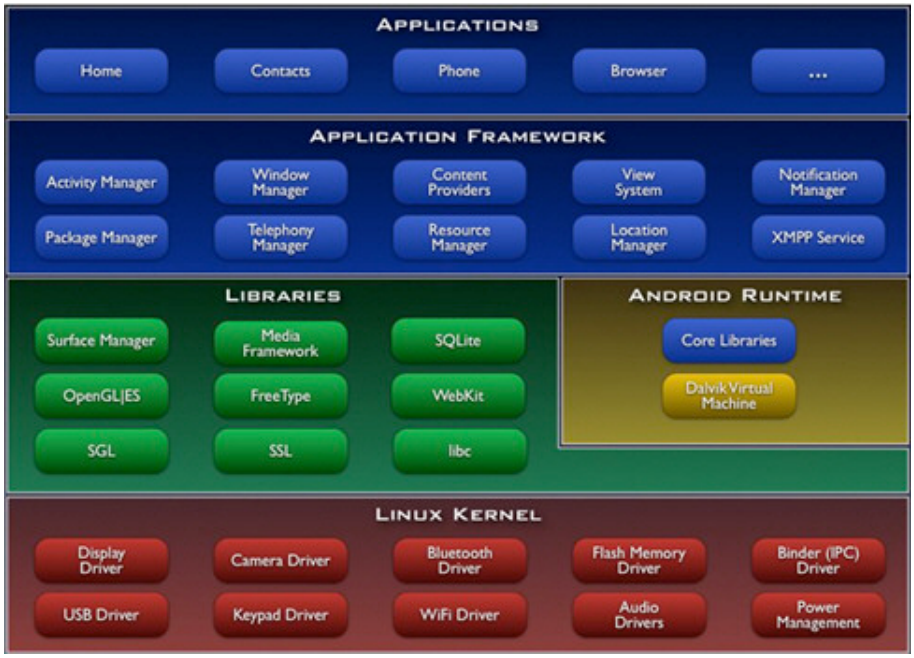

Fig. 2.1. Android architecture [And11].

SM purpose is to manage the various windows in order to draw them on a buffer to be displayed through the technique of *double buffering*[2]. So no overlapping windows will be displayed in an uncoordinated way. We stress the importance of this component because one of architecture's peculiarity is based on ability to create interactive interfaces.

Scalable Graphics Library (SGL) is a library written in C++ and with *OpenGL* is the graphics engine of Android. While for the 3D graphic there is the support of OpenGL, for the 2D graphic it's used an optimized engine called SGL. It is a library used mainly by Window Manager and by Surface Manager within the process of rendering graphics.

[2] This technique uses an additional buffer to store a data block: in this way, a potential reader will have a complete data's vision (though old) rather than a partial (but up to date) data's vision created by a writer.

SQLite library implements a very compact direct transactional relational DBMS that doesn't need any configuration. Being compact is thanks to be written fully in C in order to use only few ANSI functions for management memory. The library doesn't use any separate process to operate because lives in the same application's process that uses it.

Obviously there is an integrated browser to cope the Web 2.0: it is the WebKit framework used by browsers Safari and Chrome too. It is an open source browser engine based on HTML, CSS, JavaScript and DOM technologies. It is worth noting that WebKit is not exactly a browser but a browser engine that needs to be integrated in different applications.

SSL library deals with the Secure Socket Layer management.

Libc is an implementation of standard C library optimized for Linux-based embedded devices such as Android.

C*ore library* consists of dex version of the runtime while at compile time it will need the jar (called android.jar) for the creation of the byte-code Java. Infact there isn't Java code on the device because it couldn't be interpreted by a JVM: we find just dex code executed by DVM. These libraries represent the description of wrapper components to access functionality implemented in native way.

Application Framework (AF) is a set of APIs and components useful for the execution of very important functionality in each Android application. All libraries already seen are used by set of higher-level components which constituite the AF.

Activity Manager introduces the very essential concept of "activity" for the application's development. We can imagine and conceptualize an activity to a screen view that provides the visualization and the collection of information. Generically, activity is the basic instrument through which the end user interacts with the application.

Package Manager manages the lifecycle of applications into devices exploiting information contained in XML configuration file (AndroidManifest).

Content Provider has the responsability to manage the sharing information among various processes. It works just like a shared repository with which different applications can interact entering or reading information.

Finally, *Resource Manager,* thanks to a set of simple use API that makes available, is responsible to manage all the resources which an application needs, such as different types of files and images.

2.2 Applications: structure and parts

Activity, *Services*, *Intent* and *Intent Filter, Broadcast Intent Receiver* and *Content Provider* are the main components of Android architecture. We use them in order to program and develop an application.

An activity represents application's block interacting with the user through the display and input devices available on the smartphone. Commonly an activity uses User Interface (UI) components already present, like the ones of the *android.widget* package, but it is not necessarily the rule.

Activities are probably the most popular model in Android extending the base class *android.app.Activity*. Since an activity is a fundamental brick of an Android application, we explain its lifecycle [Fig. 2.2]. First of all, it's important to emphasize the following aspect: smartphones, unlike desktop system, have limited resource and a small display and therefore it's not a good pratice to combine two or more windows of different applications so as it's not a good pratice to keep in life too many programs simultaneously. For these reasons, Android activities have an exclusivity nature.

It's possible to have multiple tasks running simultaneously, but only one activity at time can occupy the display. Activity on display is running and interacts directly with user. The others, however, are hibernated and kept hidden in background, so as to minimize the consumption of computing resource.

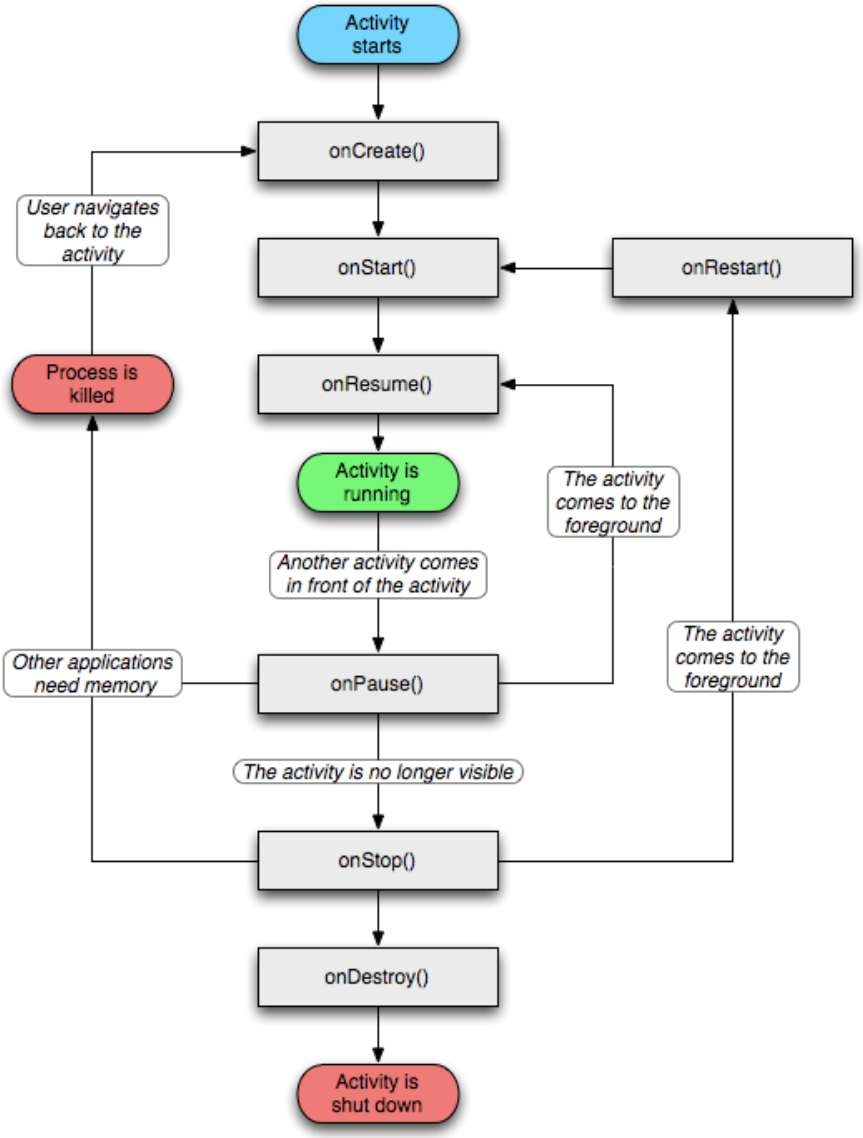

Fig. 2.2. Activity lifecycle. The flow chart shows methods call which are possible to redefine to intercept the state changes [RiP10].

Naturally user can restore an hibernated task and resume it to back it on top. Activity from which it is moving away will be hibernated and turned in background instead of the restored one. The change of

activity may also be due to an external event (i.e. due to an incoming call).

Services are running in background without having any direct interaction with user. They are usually used to perform very long tasks in background while user uses an activity to do anything else. A service is realized extending *android.app.Service* class.

Intents are useful to reuse activities in order to perform the common operations in most applications. When an application needs to perform a particular operation it will create an intent requiring the use of some resource, or component, able to satisfy the need. It is also required a mechanism that allows an application, or to its components, to declare the set of intent able to manage. This task is accomplished by intent filter.

Broadcast intent receivers are used to catch particular events through the whole system. For example, we use them when we want to perform an action while taking a picture, or when a message arrived or when the battery is going to exhaust. The class to extend is *android.content.BroadcastReceiver*.

Content providers are used to export data and information. They are a kind of communication channel among the different applications installed on the system. It is possible to create a content provider extending *android.content.ContentProvider* abstract class.

Application's architecture can be realized through components presented above paying attention to the following points :

- definition of Graphical User Interface (GUI);
- database design;
- background's operations;
- user's notification.

Android incorporates "outsourcing" concept to achieve the resource management. This concept suggests to define information outside of the place in which they are used. The advantage of this choice coming from the possibility to modify a simple configuration file rather than to recompilate code. Resources can be described from XML file or also can be binary files such as an image. *Res* folder (with its sub-

directories *drawable, layout, values, raw*) represents the structure of the resources of a project.

A final observation concerns configuration file *AndroidManifest.xml*: this file is created to offer a mechanism to describe the application to the device where it will run (not only information about GUI). This file is automatically generated in project's building phase and contains all the application's fundamental information. It specifies, for example, Android's version, main activity, needed resources and in which subfolder of *res* they are, the intent filters (if they are present), permissions that an application can exploit (such as the Internet access) and services.

Android applications are distributed as *.apk* (Android PacKage) file. Inside Android package are collected the executable files in *.dek* format, eventually associated resources and a set of descriptors which outline the package's contents.

2.3 Eclipse development: Android SDK

Google provides a development kit to exploit Android and to develop applications: it is called Android Software Development Kit (SDK) [AD11]. Although Android SDK has scripts to automate application istallation, to run the emulator and to debug code, we prefer to work within an IDE (Integrated Development Environment), namely Eclipse, in order to exploit all its facilities. This choice is supported by Android Development Tools (ADT): it's a plug-in released by Google to take advantage of one of the most widely used IDE (with NetBeans) in the world of Java programming.

After installing this plug-in onto Eclipse and expliciting the link with Android SDK downloaded, we have created an Android keystore in order to sign all the applications and to avoid exporting them in an unsigned way.

Fig. 2.3. Android emulator.

Last step before making an Android project concerns the management of AVD.

Development kit includes an emulator [Fig. 2.3] that allows us to test our applications on desktop before to export and to install them onto a real mobile device Android embedded (we work with an HTC Desire A8181). First, we learn to interact with this emulator to build applications. The first move concerns Android Virtual Device (AVD): we can create and configure as many as virtual devices on desktop we want. Through the interface made available by ADT in Eclipse we can configure our virtual smartphone defining name, target (i.e. Android version to use), the presence or not of a SD card (virtual, of course) and finally the skin (i.e. information about device's display resolution).

Now we're ready to create an Android project through a wizard just like the usually one used to create, for example, a Java project.

PART II

TuCSoN's porting over Android

3 Mobile TuCSoN

3.1 The reasons for porting

In this second part we combine two strands of our work to give birth a new item at APICe laboratory. This two aspects are, on one hand, the study and redesign of TuCSoN infrastructure for a client release and, on the other hand, the attention to new mobile technologies.

Mobile TuCSoN [Fig. 3.1] is the name of that particular release (version 1.9.2) of infrastructure leveraging o.s. Android resources; it offers to agent applications on a smartphone the opportunity to cooperate and to interact with other distributed agents.

Porting operation aims to mark the features of TuCSoN infrastructure within the new distributed intelligent systems scenarios; we show how can be simple to exploit TuCSoN's facilities from a mobile device Android embedded.

Other reasons are related to the desire of bringing agents paradigm onto the very common mobile devices. The mobility idea is concretely inherent in a smartphone and that's why it well adequate to host the distributed nature of modern multiagent systems.

The main feature of multiagent systems is to deal with a well-know management of the unpredictability and of the dynamism of the environment: for this reason multiagent systems are perfect to get stuck into Internet scenarios. The lack of a proper global state in Internet is well addressed exploiting agent's flexibility and autonomy.

In the open distributed systems so defined, it is quite clear that the management of heterogeneous environments and of possible decentralized control forms is a key obstacle to be overcome [We99].

Fig. 3.1. Mobile TuCSoN's logo.

TuCSoN infrastructure deals with the decentralized control forms issue while, as regards to the distribution and openess aspect of multiagent systems, we study how to exploit any means which could host a potential agent.

With "*porting*" we refer to those operations through which a application, originally developed for a specific platform, it's modified so well as to be used on another platform. Usually these operations are not trivial, especially when they occur among very different platforms at an architectural level.

TuCSoN's porting over Android offers the possibility of a totally new perspective to look at multiagent systems: no longer confined in a desktop, the agents will form the new MAS where the "freedom" to move with us, inside our jacket pocket, will become one of their best main feature.

The introduction of concepts, still very abstract and unfamiliar, like intelligent entity and social intelligence within mobile devices which are, instead, quite familiar and well known thanks to their commercial deployment, it represents an interesting challange. Therefore, the union of a large circulation device like a smartphone with an innovative laboratory technology based on MAS is the final theoretical

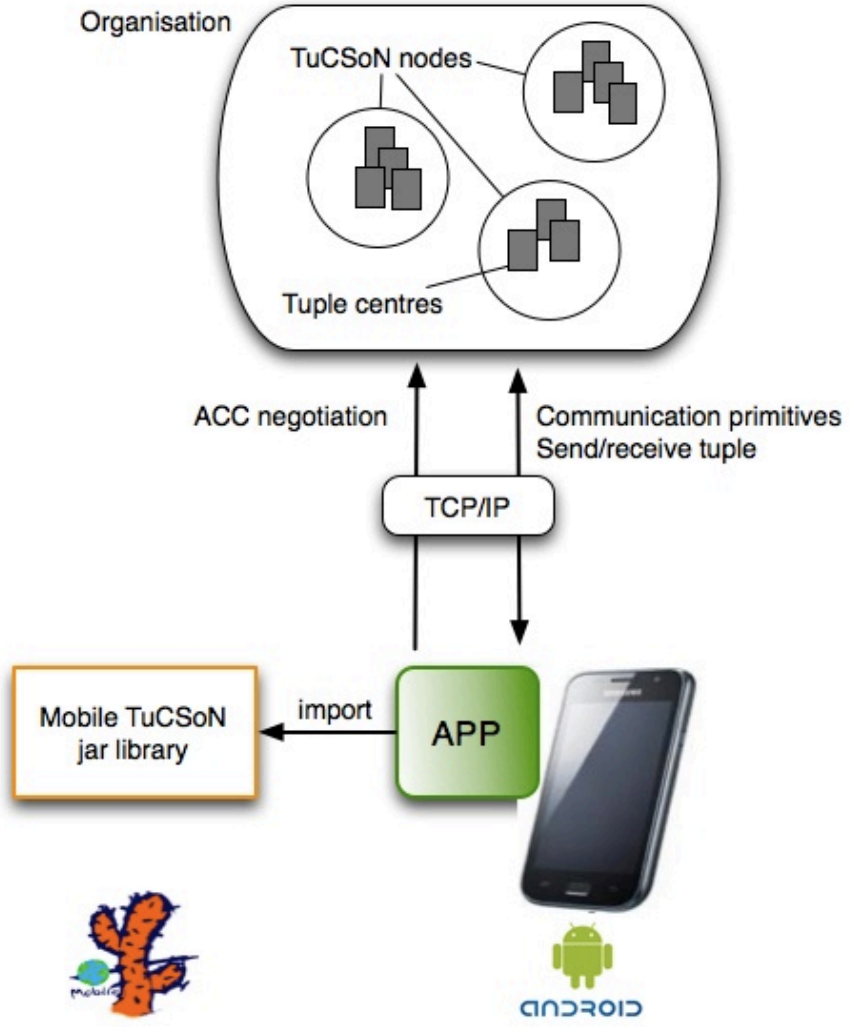

Fig. 3.2 Mobile TuCSoN utilization.

reason of our work. We think this union may have great significance conceptual and social too.

From a technical point of view, porting operation is based on the main assumption concerning the utilization of infrastructure: through a mobile device we just want to interact with the tuple centres discarding the possibility to install a TuCSoN node on a smartphone. In the latter scenario, the necessary resources and the nature of a

TuCSoN node they risk to contradict the innovative features characterizing a mobile device. Infact, a TuCSoN node must always be active during its activity within a MAS, it must reside on a “known” node and it must have a certain computing capabilities to be remotely reachable; on the other hand, a mobile device can be into different networks depending on its physical position and therefore its “dynamic” IP address would become more difficult to manage. Moreover, taking count mobile device nature, it is not useful to allocate most of its computing resources for a background service (since, as we have seen, in Android only one task at time can interact with the user) designed to remote agents.

In short, we are going to use an Android embedded smartphone to host Mobile TuCSoN version without distorting the very essence of the device: we just want to exploit the computing power available to enhance the possible locations of an agent within a MAS.

3.2 The requirements

Our porting operations to release Mobile TuCSoN version have involved an infrastructure Java written and an o.s. that contains almost all the standard Java[1] libraries.

Although this aspect represents a good starting point, it was not enough to make porting immediately.

In the first part of thesis we focused on steps to reach a client release: really, these steps were made in parallel with the definition of a possible Android application able to interact with the infrastructure.

Aware that we can take advantage of the Java libraries available by o.s., for our very first importation of TuCSoN 1.9.1 on Android we had used the full version of the infrastructure; in this way, we first try

[1] Except for *Abstract Window Toolkit (AWT)* and *Swing* libraries: infact, GUI definition is a key aspect in Android architecture and that’s why it is handled through ad-hoc libraries.

to understand which aspects to polish and which architecture to build upon knowledge basis of the two technologies.

We created an Android activity containing all the TuCSoN packages, the external libraries (util and tuprolog-1.4) and, at a first time, the configuration file tucson_conf. properties too. To manage this external resource we used the subfolder *raw* of the project's resource tree structure: within this folder, any external resource is accesible through an inputstream.

However, as the client release evolution provides the exclusion of management of the data contained in that file, in the same way we abandoned this configuration's issue on Android.

To perform at any time the updates of the client release onto Android device, we managed the different permits that a particular Android application may have. For safety (and cost!) Google o.s. prohibits, on default, to an application to exploit Internet: however, through the file manifest.xml we have set manually this permit, in order to be sure that our application could have free access to the network.

3.3 Which TuCSoN for Android

First attempts of porting are concerning version 1.4.5, as mentioned in TuCSoN chapter.

This first step allowed us addressing all the issues mentioned in the previous paragraph, except for management file tucson_ conf.properties (it's absent in version 1.4.5); so we take familiarity with the construction of an Android activity able to interact with infrastructure.

We review all the examples in the guide [ALab06] and we start to test the actual possibilities to use of infrastructure through a mobile device. Although version 1.4.5 is made up of the same essential parts

already seen for 1.9.1 version, we elect this last version for porting operation.

Essentially, this choise is based on an architectural aspect: the 1.4.5 version is rigidly connected with ReSpecT technology.

Infact, we can't disengage TuCSoN 1.4.5 from the logical language used and this feature, within a scenario even more open and extensible offered by use of mobile device, it could represent a serious obstacle as well as conceptual also implementative.

ReSpecT re-engineering and in addition TuCSoN redesign are the pratical causes of our choice: new scenarios, where we are going to, require a more dynamic TuCSoN version, such as the 1.9.1 one.

Undoubtedly, the cornerstone of 1.9.1 version is represented by the solution offered to the problem of languages separation from infrastructure. Although TuCSoN and ReSpecT technologies, and their related tasks, they have been conceived separately since their first design, only in 1.9.1 version this concept is realized in a more clearly and manageable way.

Version 1.9.1 has also maintained all other features which we wanted to exploit on client side: i.e. high degree of openness, precise definition of standard interfaces, interoperable nature, extensibility and scalability.

3.4 Proposed architecture

3.4.1 Design architecture

Mobile TuCSoN aims to exploit a mobile device environment to interact with node side of TuCSoN infrastructure: a "mobile" agent is so capable to interact and to communicate through tuple centres with others distributed agents. Since Mobile TuCSoN is addressed to mobile device, it should be light-weight and efficient.

Our proposed architecture is based on all already discussed points which we want to summarize:

- separation from ReSpecT;
- consequently client release;
- improvements and rationalization of client release.

On one hand we adopt client release derived from TuCSoN 1.9.1, on the other hand we benefit of the possibility to import java library on an Android project.

Architectural core of Mobile TuCSoN is the delegation of management of logic language on node side. So, agent side, we don't care about the "real" construction and definition of a tuple centre or an agent id.

We delegate negotiation and construction of an agent id and of a tuple centre id to node side. Although we don't have an onthology specifying the identificator of a tuple centre or of an agent, we suppose that an agent should know how to explicit the correct id form. The correct syntax for an agent id deals with any string, meanwhile for a tuple centre id the correct syntax deals with this template: *"tuple_centre_name@ip_address"*; if *ip_address* miss (and symbol "*@"* too) then infrastructure considers the *localhost* location for that tuple centre.

Delegation of management of logic language is so handled: an agent specifies its own id and the tuple centre id with which it want to interact; tuprolog-1.4 library offers the basic elements to manipulate and to create an id [Appendix A]; on agent side, tuprolog-1.4 library also verifies the correct syntax of each id through an exceptions management; the id so created is sent by AgentContextStub in the form of Java Object; on node side, AgentContextSkeleton will receive all the elements to create the relative logic structures using ReSpect (i.e. the current logic language embedded). In this scenario agent doesn't know the actual logic language used on node side.

This kind of delegation ensures separation between the require operations and the language to achieve them.

Separation from ReSpecT language (to reach a client side TuCSoN release) ensures much more flexibility and openness to TuCSoN infrastructure.

A possible extension and development of logic language will deal only with node side infrastructure; clearly, it is necessary to maintain the id syntax just seen to guarantee infrastructure integrity. This possibility represents a great advantage, for example, in a very large MAS: only node side infrastructure will be modified while all client agents don't care about this infrastructure's upgrade.

3.4.2 Detailed architecture

All packages summarized in chapter I (Fig. 1.3) are involved into *MobileTucson.jar*: in only 66KB we bring with us the possibility to enjoy TuCSoN infrastructure from a smartphone in a simple way.

Fig. 3.2 shows the structure of an Android project (it will be part of case study in the following chapter) in order to illustrate how it is composed an Android application exploiting Mobile TuCSoN.

In *src* folder we create package *mobile.moss* containing Java file of our application: it encloses Java code of the application, i.e. the code about GUI and the code about interaction with TuCSoN infrastructure.

In order to achieve flexibility, openness and efficiency we recognize the usual section concerning with external libraries; these are libraries used by application, such as Android 2.2 (just like smartphone Android version used for porting experiments) and external libraries: *MobileTucson.jar* and *tuprolog-1.4.jar*. About the latter library we have already argued in first chapter.

Now, we discuss the meaning of each package of MobileTucson library in an Android project context in relation to requirements of proposed design architecture.

Correct syntax requirement is achieved by prolog engine while all basic bricks to manage tuples are located in *alice.logictuple* package; it contains all the elements to form and to write a logic tuple under the

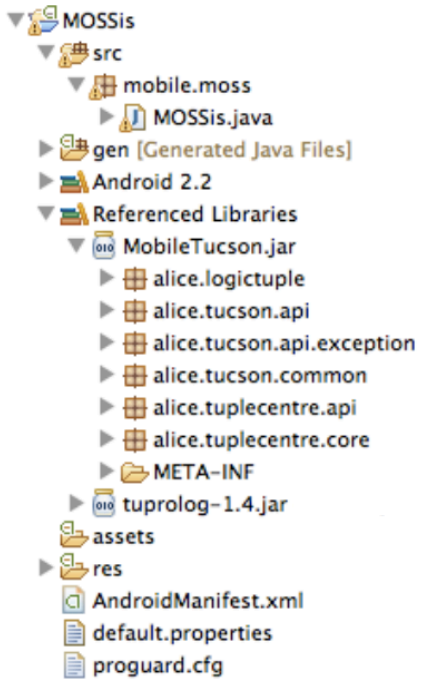

Fig. 3.1. Android project using Mobile TuCSoN.

prolog engine control; a mobile agent, infact, must be able to define, to insert or to retire (defining a tuple template) a logic tuple from/to a tuple centre.

Although Mobile TuCSoN release is characterized by lightness and by resource's rationalization, through *alice.tucson.api* package we ensure the real communication among infrastructure and agents. Infact, *alice.tucson.api* contains all the api useful to interact with TuCSoN infrastructure; among them we underline those api necessary to define an agent id and a tuple centre id and the AgentContextStub which concretely communicates with infrastructure adopting the delegation technique, above illustrated, which is the core of Mobile TuCSoN. In short, *alice.tucson.api* contains all indispensable operations which always an agent should to be aware.

Alice.tucson.api.exception contains all the exceptions useful to understand the wrong behaviours of the system in failure cases.

Alice.tucson.common, still maintaining its function, contains all the structures and all the protocol information useful to allow communication between mobile agent and node side. Nevertheless delegation's introduction, *alice.tucson.common* information are still necessary in order to guarantee uniformity to communications among agents and infrastructure.

Alice.tuplecentre.api defines all the interfaces of the structures used to interact with a tuple centre.

Alice.tuplecentre.core defines all the interfaces of the operation and behaviours used to interact with a tuple centre.

Clearly, these interfaces packages maintain their function in Mobile TuCSoN release because they represent the good basis for a well-design infrastructure in order to understand and to manage its facilities.

Separation from ReSpect and realization of a client release lead to the same results obtained in chapter I. Just like happened in client release's steps, in an Android project using Mobile TuCSoN all packages concerning Resolver, ReSpecT and TuCSoN services disappear.

Then, *gen* folder contains auto-generated file describing features application; in *res* folder we find all the resources related with the project (i.e. icons, strings); finally there is *AndroidManifest.xml* file seen yet.

3.5 Experimental results

Porting work follows sequential steps just like the release operations to obtain client side TuCSoN release.

We adopt a similar workflow for these two reasearch works: requirement and theoretic analysis, validation of a proposed solution, testing of such solution.

Every step of joint work of TuCSoN reengineering and porting operations is composed in three parts: first, we evaluate, on agent node, the theoretical meaning of a particular TuCSoN aspect, TuCSoN behaviour and TuCSoN package; then we try to fix or to remove some characteristic which is expected only node side; finally, when an agent node version of that particular aspect is ready, we test the operation through an Android application.

Through this steps and thanks to exceptions management, we are sure to control troubles coming from client version of TuCSoN or Android application.

Infact, in a porting work like that, one of main issue is represented by recognition of problem's nature. Android architecture enables us to import Java libraries and to manage them in a conventional way, just like an usual Java project within Eclipse environment.

Android o.s. exceptions together with Java exceptions represent the key aspect to face this issue. We experiment that joint management of these exception on Android mobile device is possible.

Moreover, development kit offered by Google speeds these work flow thanks to a simulator (chapter II) able to exploit a tuple centre both on local (on the desktop used for development) or on remote TuCSoN node.

Every application tested by simulator has worked fine on smartphone too. For this reason we avoid to export an Android project in a *apk* file whenever we wanted to test a particular application behaviour.

We highlight simulator's failure only when we want to exploit Android system resources (like LocationManager) related to specific services; for example, a simulator is not able to provide a GPS or network service (these are aspects useful for case study in next chapter). To test all possible Android services' utilization, we must to export the project in a *apk* file and to test it on a real smartphone.

The possibility to exploit TuCSoN infrastructure from an Android application of the order of 100KB well represents the result of our

work: i.e. to enjoy TuCSoN infrastructure from a smartphone in a very simple way through a light-weight Android application.

Our experimental results confirm all theoretical considerations about the possibility to have a client release, to enjoy TuCSoN in a separate way from ReSpecT and to interact with infrastructure only through basic operations.

No automatic test plan we adopt in porting operation; all tests we make are based on required functional skills of infrastructure. Testing phase is composed by functional examinations of basic coordination and communication primitives.

Through porting work we prove our goal is achievable: TuCSoN functionalities are all exploitable from an Android application importing Mobile TuCSoN library.

3.6 Which benefits

A light-weight Android application together with the light-weight MobileTucson library enclose all possible scenarios coming from interaction with TuCSoN infrastructure.

The well-known advantages of TuCSoN are now all exploitable in many different kind of distributed systems. Similarly, all well-known agent features [Od02] are now really testable: for example, we focus on the social aspect of an agent.

The idea to exploit an agent running on a smartphone widens distributed systems horizons. An agent-based Android application represents now a new component in a MAS society. We can defne a society like an open group of agents glued by some social tasks, which they can achieve collectively by exploiting some shared coordination artifacts, tuple centre in this case, embedding coordination laws and social norms [ROD03].

For these reasons we attribute a social significance at an agent running on a mobile device because it can compose a new kind of MAS in which agents are even more distributed.

It is rather clearly that agent's role on a smartphone is well suited on the basis of all definitions already seen. Ideas of pro-activeness, of sociality and above all of mobility are now exploitable and testable by a mobile agent on a smartphone.

TuCSoN's porting over Android carries with it all possibilities and all advantages offered by infrastructure.

4 Case studies

4.1 MOSS application

MOSS (Management Of Science Informer) application is first case study: its spirit is to show Mobile TuCSoN possibilities through a possible real scenario.

This Android application is conceived to exploit TuCSoN 1.9.2 and to demonstrate how it can be simple to use TuCSoN infrastructure from a mobile device.

MOSS aims to manage both job activities of science informers and pharmaceutical manager. Infact, we provide two MOSS versions, MOSSis and MOSSmp [Fig. 4.1]: the former is designed for science informator, the latter for pharmaceutical manager. Differences between these two applications are related to real job need of an informer or a manager: their works determine a different way to approach with TuCSoN infrastructure.

Science informer purpose is to know pharmaceutical stock availability of a specific city where he is going to work; he can consult a specific pharmaceutical type availability (choosing among general medicine, pediatrics, surgery, orthopedics and for each of these medical categories, a specific type too) in order to organize his work plan in that particular city. Science informer is a pharmaceutical consumer because he pulls out from stock all medicines which he wants to propose.

On the other hand, a pharmaceutical manager is a kind of controller able to check pharmaceutical stock availability. He’s a pharmaceutical producer because he’s able to supply a specific stock devoid of some pharmaceutical.

Fig. 4.1. MOSSis (left) and MOSSmp (right) icon.

Every stock, represented by a tuple centre, is identified by a name according to this template: *mosscity_*name*.* In this case study, we suppose to use one IP to host all tuple centres: so, location id (in the form of *"@ip_address")* will be the same for each tuple centre.

Shared information, among informers and managers, concerns about presence or not of a particular pharmaceutical in a specific city stock. Pharmaceutical information, in logic tuple form, follows this template: *medical_cathegory(subtype)*.

Thanks to ReSpecT language, every city stock (i.e. tuple centre) is programmable according to a particular need. For example we can program that a specific tuple insertion will cause the deletion or insertion of another logic tuple related to the previous. Obviously, every behaviour is concerned with particular domain of a specific city stock: a commercial policy of a city can be different than a policy of another city. ReSpecT is able to manage any possible situations concerning with a certain domain or situation.

MOSS application guarantees coordination and communication among these participants by exploiting TuCSoN tuple centres [Fig. 4.2].

Informers and managers coordinate their actions according to tuple centres content: every informer/manager's operation depends on presence or not of a particular information (logic tuple) in a specific tuple centre. So, tuple centres provide the means for communication and coordination managing shared information.

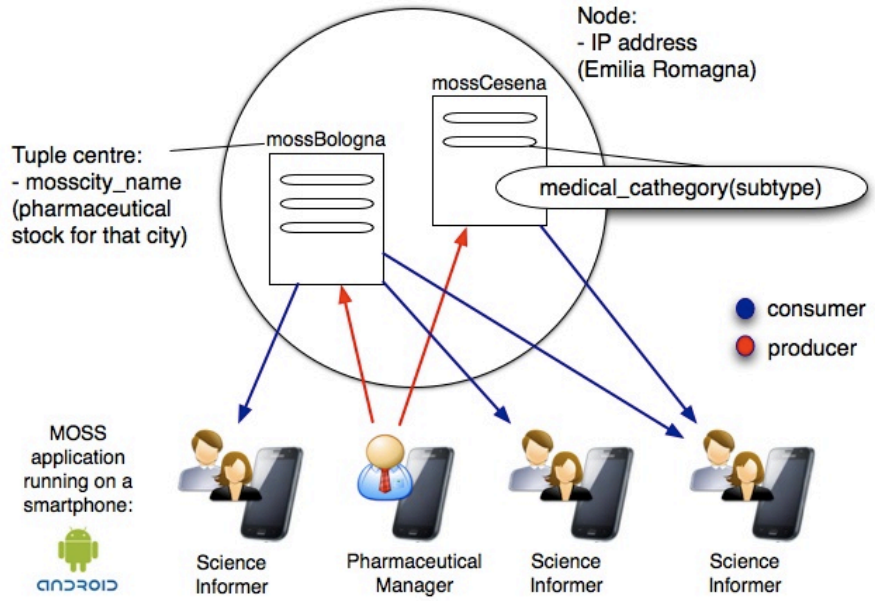

Fig. 4.2. MOSS application functionality.

4.2 How does it work

MOSS application's goal is to satisfy all requirements listed above. In its "is" or "mp" version, this light-weight Android application works using MobileTucson and tuprolog-1.4 libraries.

Through an ad-hoc GUI, MOSS collects all useful information to interact with TuCSoN tuple centres. User (informer or manager) inserts in a set of TextView all required information.

First information is about informer/manager code: we suppose every worker has an identify number; this code is used to build agent id interacting with infrastructure, in order to trace the person responsible for each action.

A TextView is used to collect information about city: user can write a city name or can detect it thanks to a *"Find city"* button. This button uses Android LocationManager which exploits systems resources

city name is used to choose the correct tuple centre with which user decides to interact.

Through two spinners user can select medical area (general medicine, pediatrics, surgery, orthopedics) and subtype for that particular area (generic, painkiller, antiinflammatory, antibiotic). This information deals with construction of logic tuples in order to check, through a *rdp* operation (not blocking read), the presence or not of that particular tuple (i.e. that particular pharmaceutical) in specified tuple centre (i.e. city stock).

"Check" button's role is to verify the presence or not of that particular tuple in specified tuple centre. When user press this button, application collects all information inserted and it builds agent id, tuple centre name, tuple and it sends an *rdp* request to TuCSoN node hosting that tuple centre.

A kind of semaphore shows us the results of *rdp* request: red light means that tuple centre doesn't contain that particular pharmaceutical, while green light means that tuple centre contains wanted pharmaceutical.

All these application's parts work identically both in MOSSis and in MOSSmp: now, depending on obtained results (coming from *rdp* request), MOSSis and MOSSmp act in different ways.

MOSSis aims to retrieve tuples from tuple centres: in red light case, application can't retrieve any tuple, while in green light case, user confirms to retrieve that particular pharmaceutical through *"Request Confirmation"* button. The button pression means that user uses that item and, in logical term, that particular tuple will be removed from tuple centre.

MOSSmp's goal is to insert tuples in tuple centres: in red light case, manager knows tuple's lack and he can decide to insert that particular tuple through *"Insertion Confirmation"* button; in green light case, manager knows tuple's presence and can both decide to insert or not another tuple of that kind (this decision is related to a possible marketing strategy in order to sponsor a particular pharmaceutical rather than another one).

4.3 Structure and GUI

MOSS structure follows project's view showed in Fig. 3.1. Both MOSSis and MOSSmp exploit MobileTucson.jar and tuprolog-1.4.jar libraries already discussed.

MOSS application is based on an Android activity: method *onCreate* ensures correct definition and creation of GUI while some buttons ensure correct operation of application in order to communicate with TuCSoN infrastructure.

Fig. 4.3 (a) shows MOSSis GUI through Android emulator used for our experiments.

Fig. 4.3 (b) and Fig. 4.3 (c) show two MOSSis spinners useful to choose pharmaceutical type and pharmaceutical subtype respectively.

Fig. 4.4 shows a successful request for a particular pharmaceutical (pediatrics, painkiller).

Fig. 4.5 shows a MOSSmp failure request for a particular pharmaceutical (orthopedics, antiinflammatory).

Latest figures, Fig. 4.6 (a), (b) and (c), show related tuple centre (mossBologna) and its content. It is reached thanks to TuCSoN Inspector tool.

As we can see, tuple centre *mossBologna* contains *pediatrics (painkiller)* but not contains *orthopedics(antiinflammatory)*.

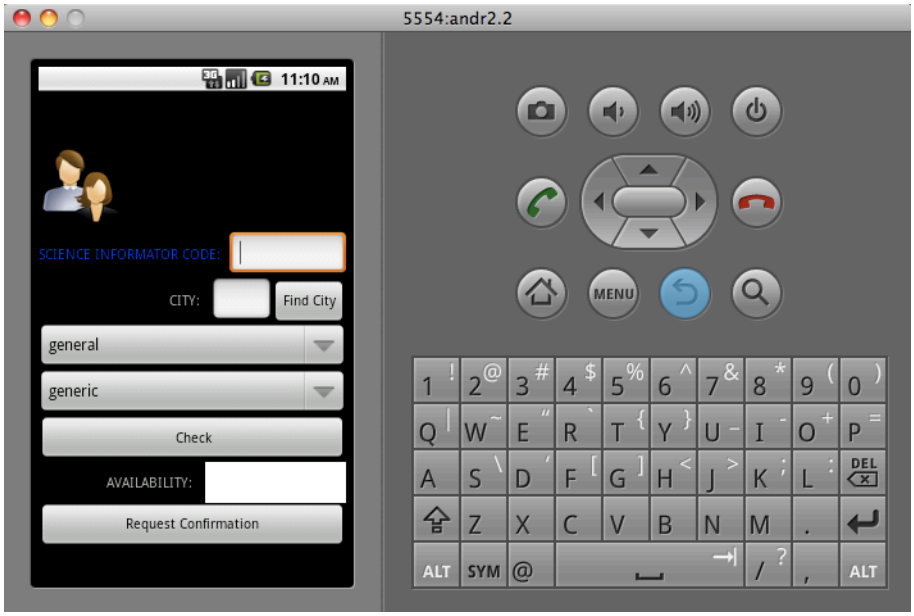

Fig. 4.3 (a). MOSSis: GUI.

Fig. 4.3 (b). MOSSis: pharmaceutical type spinner.

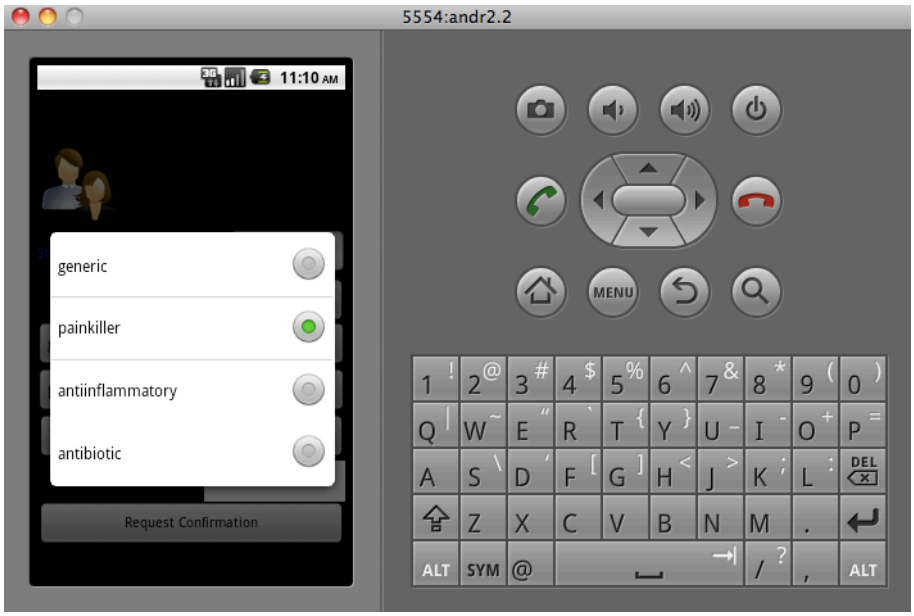

Fig. 4.3 (c). MOSSis: pharmaceutical subtype spinner.

Fig. 4.4. MOSSis: successful request for pediatrics(painkiller).

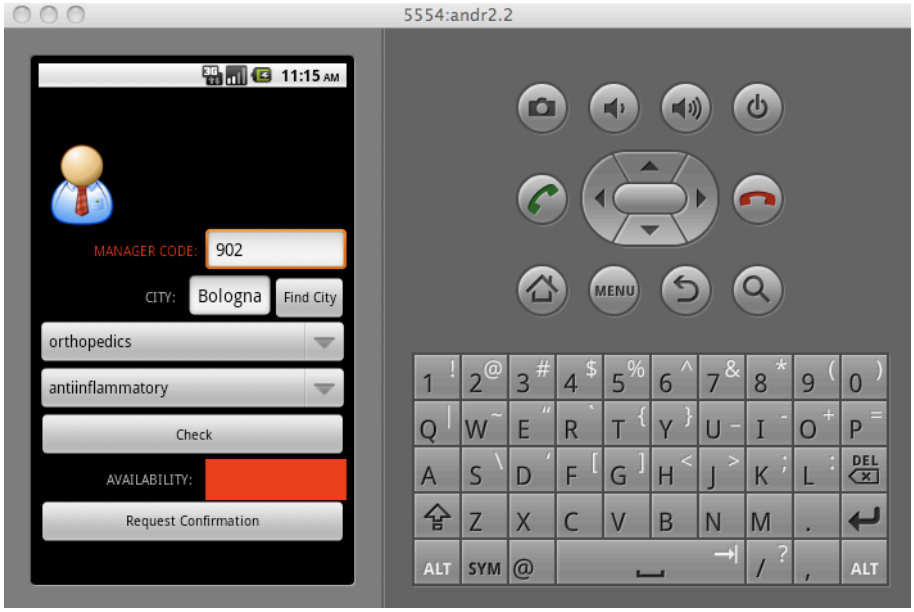

Fig. 4.5. MOSSmp: failure request for orthopedics(antiinflammatory).

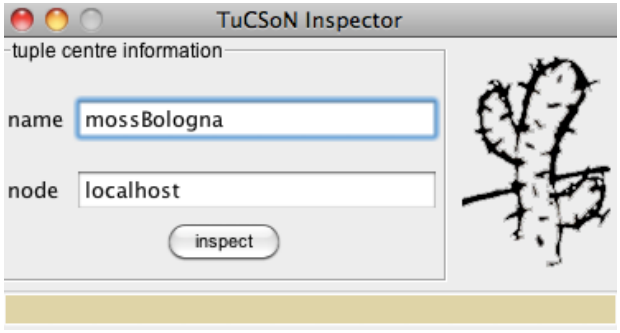

Fig. 4.6 (a). TuCSoN Inspector (I)

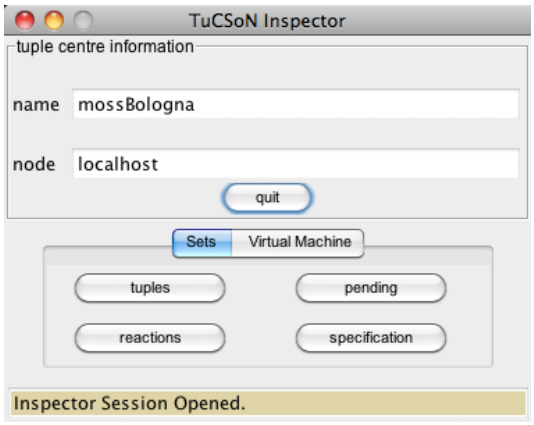

Fig. 4.6 (b). TuCSoN Inspector (II)

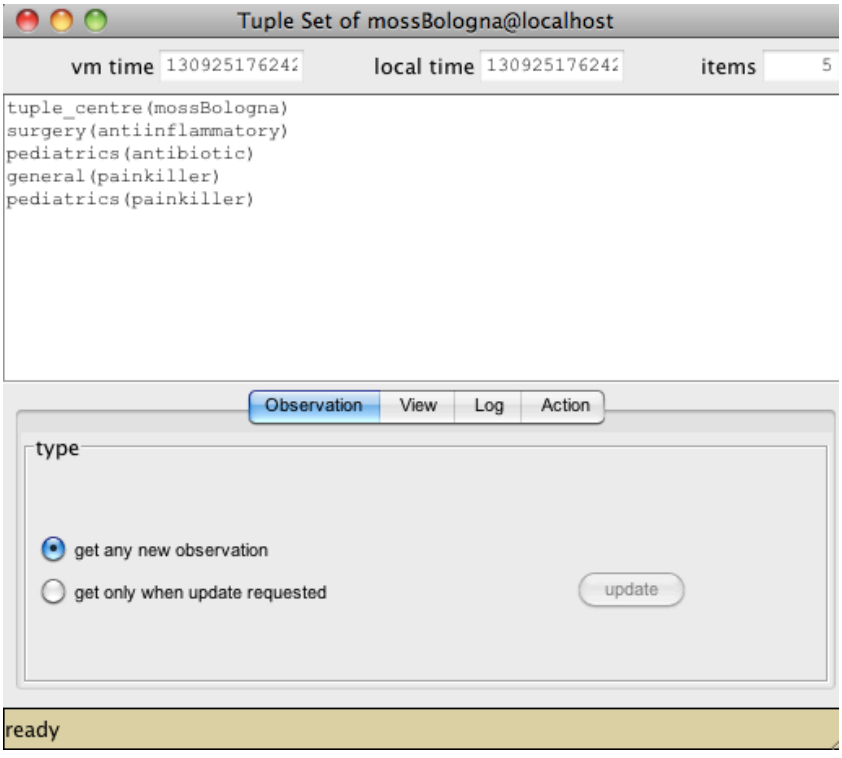

Fig. 4.6 (c). TuCSoN Inspector (III): tuple set of mossBologna.

4.4 Johannes Kepler University project

4.4.1 SAPERE context

Second case study is represented by a collaborative work with Johannes Kepler University (JKU), Linz (Austria), in SAPERE[1] project context.

SAPERE takes inspiration from natural ecosystems and starts from the consideration that the dynamics and decentralization of future pervasive networks will make it suitable to model the overall world of services, data and devices as a sort of distributed computational ecosystem [SAP11].

SAPERE considers modelling and architecting a pervasive service environment as a non-layered spatial substrate, laid above the actual network infrastructure.

Around SAPERE approach it's possible to distinguish studies related to:

- a model for components and the associated methodology (WP1 "Model and Methodology");
- self-organization in a distributed network space (WP2 "Structure and Space");
- knowledge management for situation identification and future-awareness (WP3 "Knowledge and Time");
- defining and implementing an innovative middleware architecture to support the model and algorithms (WP4 "Infrastructure");
- testing and evluating on selected use cases in the area of pervasive services (WP5 "Applications").

[1] Self-Aware Pervasive Service Ecosystems.

4.4.2 Live sensor data

JKU's people, Alois Ferscha, Bernhard Wally and Sascha Maschek, are preparing a runtime kernel to emulate SAPERE functionality. This project is located in WP4 study area of SAPERE approach and its based on a live sensor data realization.

The infrastructure contains a tuple space implementation as coordination model to be easily exchangeable with a later SAPERE space implementation.

LSA[2]-like data structure can be injected into those tuple spaces while a reasoning engine evaluates local LSAs and potentially fires actions [Fe11].

Project architecture [Fig. 4.7] bases on tuple space, sensor and other services and a reasoning engine. Display node is just like as OSGi[3] container.

Tuple space management is entrusted to TuCSoN for data sharing and inter-process communication (IPC).

Demonstrator infrastructure is really tested first thanks to virtual sensors, and then through real sensors (Broadcom BCM4751 and YAMAHA MS-3C). Hardware used is an Android smartphone with the appropiate built-in sensors: location sensor (GPS), orientation sensor (compass) and motion sensor (accelerometer). Software required is an Android app installed on smartphone, an Android Debug Bridge (ADB) and a client server architecture (smartphone like a client, computer like a server).

JKU's project purpose is making public dislays location-, orientation-, and motion-aware.

Hardware solution is so organized: laptops are used as public displays which are interconnected via Wi-Fi; smartphones attached to public displays providing live sensor data; user presence detection is so realized [Fig. 4.8].

---

[2] Live Semantic Annotations.

[3] Open Services Gateway initiative.

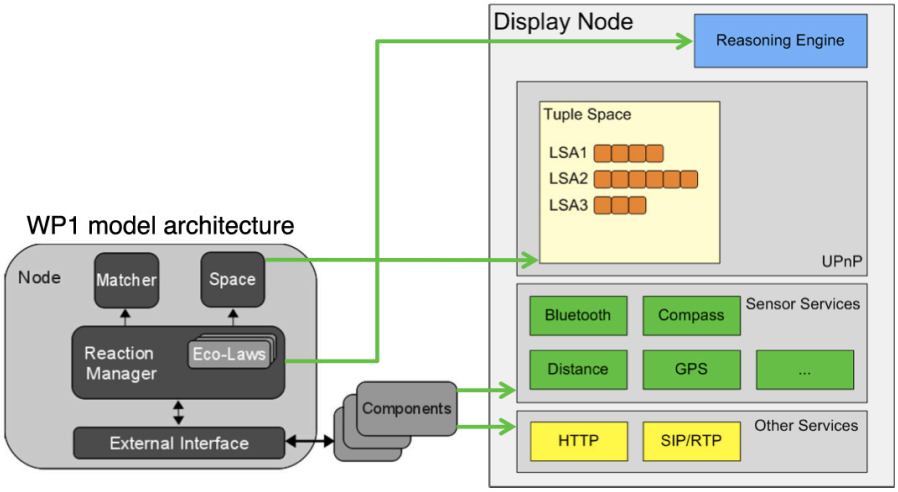

Fig. 4.7. JKU project architecture [Fe11].

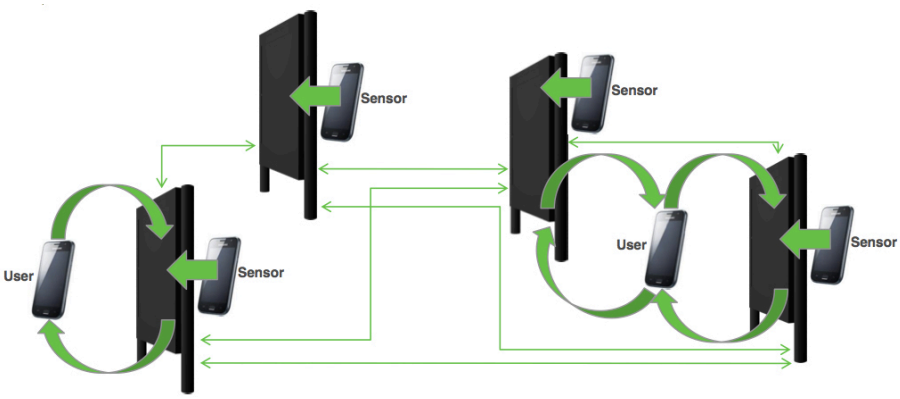

Fig. 4.8. Hardware solution [Fe11].

4.4.3 Collaborative work

Collaborative work is conceived to obtain a stable common TuCSoN release useful for both our and their research. Since live sensor data project uses smartphones Android embedded to interact with TuCSoN, we propose to introduce our Mobile TuCSoN release.

First steps of collaborative work is focused on some bugs resolution related to whole TuCSoN infrastructure. For example, different identifiers for tuple centres and remote port specification are two bugs which are needed to be resolved.

First bug is caused by a wrong initialization of CLIAgent tool: this tool, through a command line, can interact with tuple centres. Creating every time a new context useful to interact with tuple centre, we can both use ‘localhost’ or ‘IP address’ to specify a tuple centre.

Bug about remote port specification is solved by a careful analysis of string manipulation in AgentContextStub file.

It’s pretty clear that many scenarios may coming out from tests like that in order to achieve a complete bug fixed release. We consider very important collaborative works like that because they represent a valid contribute to projects development.

Conclusions and future work

Thesis shows how MAS can be really open: agents distribution takes relevance in our work in order to exploit TuCSoN infrastructure from a smartphone. Distribution assumes a more marked relevance as well as TuCSoN's role becomes indispensable.

Separation of languages from infrastructures represents another important conceptual basis coming out our work. We analyze this aspect to achieve a client release. Furthermore TuCSoN re-engineering shows the importance to distinguish tasks and behaviours related to node side or client side.

Client release (so called Mobile TuCSoN, version 1.9.2) gives us possibility to appreciate the concrete TuCSoN abilities exploited by an agent. Thesis focuses on agents on Android mobile devices because TuCSoN's porting over Android smartphone is our final goal. Client release offers the opportunity to address new scenarios in which agents don't need to bring all whole TuCSoN infrastructure but just a functional release. Although our goal is Android oriented, is pretty clear that client release could be reuse to face new technologies.

That aspect represents an interesting starting point for future works releasing a new client version for other embedded smartphone.

Through TuCSoN's re-engineering and porting we deduce some theoretical and pratical considerations regarding architectural, structural and engineering aspects.

About architectural and engineering issues we underline the importance of a well-designed project. Infact, we expect that an infrastructure is modular and scalable, that allows to separate and to select only interesting aspects for our needs. TuCSoN client release demonstrates this goal is reachable starting from a well-designed in-

frastructure. We consider a scenario in which a mobile agent simply would exploit TuCSoN infrastructure without installing a TuCSoN node on a smartphone.

The latter perspective represents an open point for future developments: when computing skills of smartphones or different requirements will lead us to install a TuCSoN node on a mobile device.

About structural aspects we remark o.s. Android organization and its characteristic to manage Java code. An open source project like Android, capable to import Java libraries, offers many possibilities in order to interact with devices embedding it. No doubt features of Android application are a fundamental bridge towards thesis goal.

We suggest other two open points for future works: they are related to new tuprolog libraries and to possibility to define an onthology for agent id and tuple centre id.

We guess is very important to import new tuprolog libraries in future TuCSoN's release, expecially tuprolog-2.4 [Appendix B]. Together with a new ReSpecT version, TuCSoN will be so able to exploit all news and all bug fixs characterizing tuprolog-2.4.

A well-know onthology will be useful to client agent because it could interact with TuCSoN through well defined structures, still maintaining no information about logical language used by infrastructure. Onthology will be about the definition of agent id and tuple centre id: so we ensure that every agent knows the correct syntax to use TuCSoN's elements.

Finally, MOSS application and Live sensor data project represent two different utilization way for Mobile TuCSoN: those scenarios lead us to consider the concrete value of our research in order to be a basis for future applications.

Appendix A

A.1 Mobile TuCSoN's dependencies from tuprolog-1.4

alice.logictuple PACKAGE

Class	Dependencies
LogicTuple	alice.tuprolog.Term;
TupleArgument	alice.tuprolog.Term; alice.tuprolog.Struct; alice.tuprolog.Number;
Value	extends TupleArgument
Var	extends TupleArgument

alice.tucson.api PACKAGE

Class	Dependencies
AgentContextStub	Import alice.logictuple
TucsonAgentId	alice.tuprolog.Term; alice.tuprolog.OperatorManager
TucsonOperation	Import alice.logictuple
TucsonTupleCentreId	alice.tuprolog.Term; alice.tuprolog.Struct

alice.tucson.common PACKAGE

Class	Dependencies
TucsonMsgReply	*Import alice.logictuple*
TucsonMsgRequest	*Import alice.logictuple*

A.2 Tuprolog-1.4 library packages used

alice.tuprolog;
alice.tuprolog.lib;
alice.tuprologx.ide;
alice.tuprologx.runtime.tcp;

Appendix B

B.1 Future work and future dependencies about new tuprolog-2.4

In following tuprolog versions we find lots of novelties: since version 2.0 to 2.4, there are so many differences with version 1.4 here used.

Tuprolog 1.4, on which ReSpecT 2.2 is based on, is still wrapped on some structures, come from *java.util* library, which required many casting operations to work fine. Anchored to prolog engine of the previous releases, tuprolog 1.4 does not consider a whole number of issues which will be dealt in future.

Tuprolog 2.4 completes the bug fix process started in past versions (2.0, 2.2, 2.3). Also it copes the catching exceptions problem, the collection structures use, a fast indexing method, a new graphical suit, the merge with utilities functions of the *util.*jar library and finally it provides a prolog plugin for Eclipse.

New prolog version, still maintaining the same packages and the same basic architecture, it introduces a lot of new methods able to manage new structures adopted and it requires much more computational resources in order to benefit indexing method.

For these reasons we leave tuprolog 2.4 adoption to future work: it could be a good basis for a new ReSpecT release. For example, in our infrastructure a fast indexed method to manage prolog terms it will be superflous: it would be really useful only under a particular extreme situation, far away from our scenarios which involved prolog's terms.

A new ReSpecT language will be able to exploit all this new features offered by tuprolog 2.4 and, consequently, future TuCSoN versions will have the possibility to enjoy that new logic language.

References

[AD11] Android Developer web site. 2011. http://developer.android.com/sdk

[ALab06] APICe Lab: TuCSoN Guide. TuCSoN version: 1.4.5. 2006.

[And11] Android web site. 2011. http://www.android.com

[API11] APICe web site. 2011. http://alice.unibo.it

[Ca10] Carli, M.: Android. Guida per lo sviluppatore. Apogeo. 2010.

[COZc00] Cremonini, M., Omicini, A., Zambonelli, F.: Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach. 2000.

[COZm00] Ciancarini, P., Omicini, A., Zambonelli, F.: Multiagent system engineering: the coordination viewpoint. In Jennings, N.R., Lespèrance, Y., eds.: Intelligent Agents VI - Agent Theories, Architectures, and Languages. Volume 1767 of LNAI., Springer-Verlag, 2000.

[DOR01] Denti, E., Omicini, A., Ricci, A.: tuProlog: A Light-Weight Prolog for Internet Applications and Infrastructures. 2001.

[Fe11] Ferscha, A.: SAPERE. Consortium Meeting. University of St. Andrews, Scotland. June 22, 2011.

[FGM03] Ferber, J., Gutknecht, O., Michel, F.: From agents to organisations: an organizational view of multi-agent systems. In: 2nd International Joint Conference on Autonomous Agents and Multiagent Systems, Melbourne, Australia, ACM Press, 2003.

[NM09] Natali, A., Molesini, A.: Costruire sistemi software: dai modelli al codice. Seconda edizione. Progetto Leonardo, Bologna. 2009.

[Od02] Odell, J.: Objects and agents compared. Journal of Object Technologies. 2002.

[OHA11] Open Handset Alliance web site. 2011. http://www.openhandsetalliance.com

[OR04] Omicini, A., Ricci, A.: MAS Organization within a Coordination Infrastructure: Experiments in TuCSoN. 2004.

[OZ99] Omicini, A., Zambonelli, F.: Coordination for Internet application development. Autonomous Agents and Multi-Agent Systems. 1999.

[PO02] Parunak, H.V.D., Odell., J.: Representing social structures in uml. In Wooldridge, M., Weiβ, G., Ciancarini, P., eds.: Agent-Oriented Software Engineering II, Second International Workshop, AOSE 2001, Montreal, Canada, May 29, 2001, Revised Papers and Invited Contributions. Volume 2222 of LNCS., Spronger-Verlag, 2002.

[RiP10] Redazione Io Programmo: (eBook) Android Programming. Edizioni Master. 2010.

[ROD03] Ricci, A., Omicini, A., Denti, E.: Activity Theory as a framework for MAS coordination. In Petta, P., Tolksdorf, R., Zambonelli, F., eds.: Engineering Societies in the Agent World III. Volume 2577 of LNCS. Springer-Verlag. 2003.

[SAP11] SAPERE web site. 2011. http://www.sapere-project.eu

[SO08] Semprini, L., Omicini, A.: Riprogettazione, separazione ed integrazione delle tecnologie TuCSoN e ReSpecT per Sistemi Distribuiti. 2008.

[ZJW01] Zambonelli, F., Jennings, N.R., Wooldridge, M.: Organisational rules as an abstraction for the analysis and design of multi-agent systems. International Journal of Software Engineering and Knowledge Engineering, 2001.

[We99] Weiss, G.: Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. The MIT Press. 1999.

[WJ95] Wooldridge, M. And Jennings, N.R. Intelligent agents: Theory and practice. Knowledge Engineering Review, 1995.