A Multi-Agent Security System for Android Platform

by

Zhiyong Cheng

B.Sc., The University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

December 2012

Abstract

The Android mobile platform is fast becoming the most popular operating system for mobile devices. Although Android security is an emerging research area and there have been many commercial and research solutions made available, the resource constrained nature of mobile devices dedicates a continuous pursuit for efficiencies. In this thesis, we present the design and implementation of a multi-agent security system on the Android platform, which is built on the Foundation for Intelligent Physical Agents (FIPA) specifications [1] compliant Java Agent Development framework (JADE) [2]. A prototype system is implemented and studied. In our design, the agents in the prototype system are aware of resource constraints such as battery capacity, network bandwidth, and dynamically adjust their behaviors accordingly to achieve a balance between the resources consumption and security needs. Following an analysis and design methodology recommended by JADE [3] and Android development guidelines, the prototype system provides compatibility with other multi-agent systems and allows easy adaptations to many security scenarios. Several baseline performance measurements are adopted to measure the efficiency of the prototype system.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACL	Agent Communication Language
ADB	Android Debug Bridge
ADT	Android Development Tools
AID	Agent Identifier
AMS	Agent Management Service
AP	Agent Platform
API	Application Programming Interface
AVD	Android Virtual Devices
BLE	Bluetooth Low Energy
CPU	Central Processing Unit
DEX	Dalvik Executable
DF	Directory Facilitator
FIPA	the Foundation for Intelligent Physical Agents
GPS	Global Positioning System
GPX	GPS eXchange file
GUI	Graphical User Interface
HAXM	Hardware Accelerated Execution Manager
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
IDS	Intrusion detection system
IEEE	Institute of Electrical and Electronics Engineers
IMEI	The International Mobile Station Equipment Identity

IP	Internet Protocol
IPC	Inter Process Communication
JADE	Java Agent DEvelopment Framework
JDK	Java Development Kit
JRE	Java Runtime Environment
KML	Keyhole Markup Language file
KVM	Kernel-based Virtual Machine
LEAP	Lightweight Extensible Agent Platform
MAS	Multi Agent System
MTP	Message Transport Protocol
MTS	Message Transport Service
OHA	Open Handset Alliance
OS	Operating System
RMA	Remote Monitoring Agent
SD	Secure Digital
SDK	Software Development Kit
SL	Semantic Language
SMS	Short Message Service
UML	Unified Modeling Language
VM	Virtual Machine
VT	Virtualization Technology
XML	Extensible Markup Language

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Dr. Son Vuong, for all his help and support. I also would like to thank Dr. George Tsiknis for kindly being my second reader and his comments.

Thanks to all the past and present NIC lab members. Specially, I want to thank Billy and Wei for proof reading my thesis; Shahed and Andrew for their valuable insights; and others for their comments and suggestions.

Finally, my special thanks go to my wife for her encouragement and support.

Chapter 1: Introduction

1.1 Motivation

Originated from and developed by the Google-led Open Handset Alliance (OHA) [4], Android has gained popularity rapidly as a mobile device operating system in recent years. As of the third quarter of 2012, three out of four smart phones shipped in the world are built with Android [5]. The growth can be attributed to its open source nature [6], native JAVA development environment, active developer community and wide device manufacturers' support.

Along with its popularity, the security of Android has become a serious concern and urgent concern. The security threats are real: Malicious applications have been uploaded to Android Market (Now Google play store) [7]; Rootkit have been shipped with Android smartphones provided by many wireless service providers to track users sensitive activities such as keystrokes and SMS message contents without their knowledge [8]; and a new cluster of malware exploits SMS payment system in China to steal bank card information and make unauthorized payments [9].

Android security is an emerging research area with promising practical outcomes and many commercial security solutions are already made available in the market. Although proprietary commercial products are generally difficult to evaluate, a report [10] published by an independent IT security research company, AV-TEST Institute , shows that less than half of the Android security applications in the market are efficient in malware detection (In the report, a security application is considered as efficient if it can detect more than 65% of the malware samples).

In contrast, academia takes a more transparent and systematic approach to addressing the Android's security issues. One commonly used approach is extending the Android security framework: By modifying the Android Dalvik virtual machine code, TaintDroid [11] introduces taint tracking on sensitive data stored on Android devices; Kynoid [12] extends the approach further and provides real-time security policy enforcement for Android; Poly implemented in Apex framework [13] is an extended Android package installer which allows users to have more control over runtime constraints on Android applications. Another popular approach is porting existing Linux security systems to Android. Shabtai et al. [14] classify Android threats into five categories and recommend incorporating Linux security solutions such as SELinux [15], firewall, intrusion detection system (IDS) and context-aware access control into Android as high priority tasks; SELinux is implemented on Android later by the same group of researchers [16] and the evaluation results show significant performance degradation on CPU and memory usage; Schmidt et al. [17] recompiled various Linux tools to enhance Android security mechanisms but also faced limited resources issues. Despite both approaches improving the Android security, each has its own limitations. Google increased the release cycle of Android to twice a year recently and there is a very likely chance of more frequent updates in the future. This poses a huge challenge for the "extending Android operating system code" approach because every system update may introduce new features that are not compatible with previous system modifications. On the other hand, as Android mobile devices are usually constrained by resource limitations such as: battery powered, lower CPU frequency, limited memory and storage, simply-porting

traditional Linux security tools directly to Android without redesign or optimization is again not a feasible solution.

To overcome these issues, we attempt to classify components of a typical Android security system into two groups: the data collection components that need to be resident on Android devices at all times and interact closely with Android local resources, and the service provider components which take data collected and make recommendations based on information available. With the help of the multi-agent system framework, components can be wrapped as agents and dynamically adjust their behaviours according to the resources available and feedbacks from other agents. Agents collecting data on Android platform are fully aware of resource limitations while agents who provide services can rely on computation resources on the server to perform security analysis and make recommendations accordingly. This approach will not be affected by Android operating system upgrades since no Android system code modifications are required. Also, our approach avoids porting heavy-weight Linux security solutions directly to Android because only data collecting components will be deployed onto the device.

1.2 Thesis Contributions

The most important contribution of this thesis is the design and implementation of a multi-agent security system for the Android platform that separates data collection tasks from the security analysis process. Followings are other main contributions from this thesis:

- The design and implementation of four types of agents on Android which are self-adaptive to resource constraints such as battery capacity and network bandwidth.

These agents can take actions independently or perform commands from service agents.

- Design and implementation of three types of service agents to handle location, subscription and profile queries and requests from mobile agents on Android.
- Design and implementation of an application specific Ontology to handle communications between agents.
- Leverages light-weight Linux network analysis tools in a rooted Android environment for advanced data collecting tasks.
- Explore the possibility of extending the prototype to support legacy Linux security system.

.

1.3 Thesis Structure

This thesis is structured as follows: Chapter 2 introduces the background information for android security, multi-agent system frameworks and related research in this area. In chapter 3, we propose and analyze the infrastructure design of our system. Then, chapter 4 presents the prototype implementation of this system. Chapter 5 evaluates the efficiency of the prototype system based on several baseline performance measurements. Finally, we conclude the thesis and discuss future works in Chapter 6.

Chapter 2: Background

This chapter provides some background information on Android security model, multi-agent systems, FIPA standards and the JADE development framework.

2.1 Android security

2.1.1 Android basics

Android is a Linux-based operating system specifically designed for mobile devices such as smartphones and tablets. The Android platform is built on a layered structure of services: a Linux kernel which provides abstraction of system hardware, a middleware presented as Dalvik virtual machine and core functionalities such as web browsers and phone dialers are implemented as applications. Most android applications are developed in JAVA and compiled to byte-code in DEX format. During runtime, the Dalvik virtual machine executes the byte-code in separate process id to prevent possible compromises such as buffer overflow attacks.

Every Android application is composed of four core types of components [18]. Activity is the user interface component where a user interacts with the application; The Service component usually runs in the background even when the application itself is out of focus; The Content provider component allows sharing data between applications; The Broadcast receiver receives implicit intent messages asynchronously and processes them.

The binder IPC (Inter-process communication) mechanism in middle-ware handles all the inter-application communications. Components interact within the Android application or between applications using intent messages. Any component who wants to receive certain

intent messages must register intent filters by specifying action strings uniquely identifying the wanted intent messages.

2.1.2 Android security

Android provides layered security mechanisms [19]: At the operating system level, Dalvik virtual machine runs each Android application in a process with low-privilege user identity and by default, an application can only access files associated with its own identity. This sandbox design minimizes the effects of a compromise such as a buffer overflow attack since the exploit is limited to the application itself. The application-level Android security framework is based on permission labels. Android applications identify the permissions needed in their manifest file and the user will grant them at installation time. Once the permissions are granted, the applications cannot adjust them without reinstallation. In other words, Android permissions cannot be granted at run-time.

Interestingly, the official Android application store, Google Play Store, can be considered as another layer of Android security mechanisms. Although Google doesn't review applications before listing them in the store, these applications may be reviewed at later time. If an application is identified as malware either by user complaints, security research or any other sources, the Android security team can remotely remove the malware from users' devices [20].

Here are some common issues related to Android security framework:

- The application permission labels are so complicated that most Android users and even developers have difficulties to understand their meanings [21].

- Application developers tend to request more permissions than needed or ask permission for the wrong purpose. One example is using IMEI as a device unique identifier, which requires READ_PHONE_STATE permission.
- The complexity of Android message-passing mechanism increases attack surfaces, which lead to exploitable vulnerabilities [22].

2.2 Multi-agent system framework

2.2.1 Multi-agent system

A multi-agent system (MAS) [23] is a system composed of multiple interacting agents within a common environment. In this context, the agents are computer programs that can independently perform certain tasks and interact with other agents. Two key aspects of MAS are agents and their common environment. Multi-agent systems can be used to solve problems that are hard for a single agent to solve and applied in a wide variety of domains such as resource management, data mining, distributed control and more [24]. However, constructing a multi-agent system is a difficult task because it requires a good understanding of both traditional distributed systems and intelligent agent interactions requirements. There are many multi-agent system development frameworks and methodologies available and we must choose one that meets our requirements.

2.2.2 FIPA standards

The Foundation for Intelligent Physical Agents (FIPA) was founded as an international non-profit organization in 1996 to form a set of standards for agents and agent based systems. In June 2005, the FIPA was absorbed into the IEEE computer society as one of its standards

committees [25]. The agent management and agent communication language (ACL) specifications are the two most widely adopted FIPA standards.

An agent platform implementation should have all of the following logical components defined in FIPA agent management reference model [26]:

- *Agent platform (AP)*: This is the physical infrastructure in which all the agents will be deployed. The agent platform contains agents, FIPA agent management components described below and any other supporting software.
- *Agent:* An agent is a computational process with some service capabilities that can be published as service descriptions. Agents must have communication functionalities and communicate using FIPA ACL messages.
- *Directory Facilitator (DF):* DF is implemented as yellow page services to other agents. Agents can make their services available by registering their services with the DF. Also, agents can query the DF to find out what services are offered by other agents. DF is an optional component and is part of FIPA agent management components.
- *Agent Management System (AMS):* AMS is the "inventory and operation manager" of the AP. Every agent in the AP must register with AMS to obtain a valid AID and AMS maintains a directory of all the agents present within the AP. All the common agent operations such as creation, deletion and migration must go through AMS. A single agent platform can only have one AMS at any time. AMS is a mandatory component and also part of FIPA agent management components
- *Message Transport Service (MTS):* MTS is the default message service in the AP. It can transport FIPA ACL messages between agents within the AP or on different APs.

FIPA ACL specifications [27] defines a standard language for message encoding and semantics which is used by agents to exchange messages and describe commonly used communicative acts such as inform, request, agree, not understood and refuse. An agent fully compliant with FIPA ACL standards should be able to receive all the standard messages and at least respond with a not understood message if it is not able to process any received messages. FIPA also defined a set of interaction protocols [28] to extend the basic communicative acts to actions.

2.2.3 Java Agent DEvelopment Framework (JADE)

JADE is a software framework for developing multi-agent applications in compliance with FIPA specifications. As an open source project, the JADE system including source code and documentations is downloadable from Tilab website [29]. JADE is fully developed in JAVA and has released runtime libraries for Android platform [30]. Here are some important features JADE provides as a multi-agent application development platform:

- A FIPA-compliant agent platform: FIPA agent management components such as AMS, DF are prebuilt and automatically started with the JADE agent platform.
- Distributed agent platform: The agent platform can be split between several hosts including mobile devices and only one Java application is executed on each host. Each agent runs as a separate Java thread and communicates with each other transparently.
- A transportation mechanism facilitates sending and receiving FIPA ACL messages between agents.
- A library of FIPA interaction protocols is included.

- A set of graphic tools to support testing and debugging.
- Support for application specific ontologies and content languages.
- Support for agent mobility and more.
- JADE platform is widely adopted within the software agent development and research communities.
- The JADE project is well developed, maintained and documented.
- The ongoing support for Android platform and constantly updated Android run-time library

We choose JADE as the multi-agent development platform for implementing our system also because JADE's features presented above.

Chapter 3: System Analysis and Design

In this chapter, we first choose the methodology for guiding the analysis and design of our proposed system. Then we define a typical Android security scenario and analyze the scenario from the multi-agent system perspective. Finally, we present the design of key components in the system such as agents, agent behaviours and application specific ontology.

3.1 Methodology

Agent-oriented software system is a new research area in software engineering. Since the agents in such systems are fundamentally different from objects in object-oriented paradigm [31], typical object-oriented system design methodologies cannot be directly applied. Thus, choosing a well-defined agent-oriented development methodology is the important start point for the analysis and design of proposed system. In recent years, multiple agent-oriented software development methodologies are proposed for developing multi-agent systems. Some prominent methodologies such as Gaia [32], Tropos [33], and MaSE [34], are generic for all the major agent development frameworks and cover most the stages of system development. Since JADE is used as the underlying agent development framework for our proposed system, we choose the methodology [3] that is specifically developed for JADE. This methodology targets only at the analysis and design stages.

3.2 System Analysis

3.2.1 Scenario analysis

The proposed system should be capable of performing the following tasks:

- An Android mobile device can subscript to any service provider and the provider will send a list of nearby registered devices. The mobile device is responsible of maintaining the list and decides how to communicate or interact with nearby devices.
- An Android mobile device checks its environment such as operating system version, network type and IP address periodically and reports back to service provider. The service provider will perform necessary analysis on the data and accordingly make recommendations to the mobile device. The recommendations include but not limited to turning on firewall, activating antivirus program, using https whenever applicable and etc. The mobile device can then take actions based on the recommendations.
- An Android mobile device updates its current location and sends information to service provider with user's permission. The service provider keeps location history of each device for analysis purpose.
- A rooted android device should be able to take more proactive actions than non-rooted devices. Some possible actions include but not limited to :
    - Snapshot network states.
    - Capture network packets.
    - Intrusion detection.
    - Network forensic and etc.
- The service provider should be able to differentiate rooted and non-rooted devices and make recommendations accordingly.

Assumptions of the system:

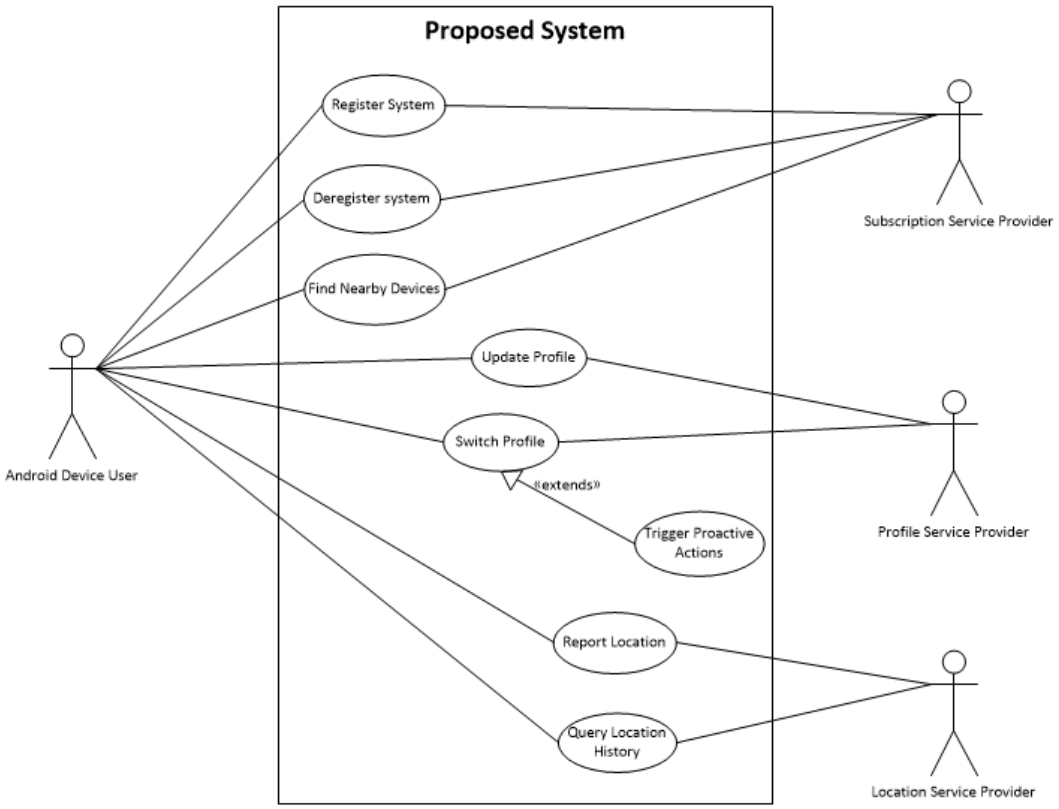
- Only focus on Android platform for mobile devices.
- Cross-platform agent mobility is not considered.

- Agent platform security is provided by JADE plugins, refer to JADE documentation for details.

3.2.2 Use case

Use cases show how a user interacts with the proposed system (Sometimes, another user or system) to achieve a special goal and are generally accepted as an efficient way to depict system functional requirements. Following the popular Unified Modeling Language (UML) specification industrial standard, the use case diagram of proposed system is shown in the following Figure 3.1:

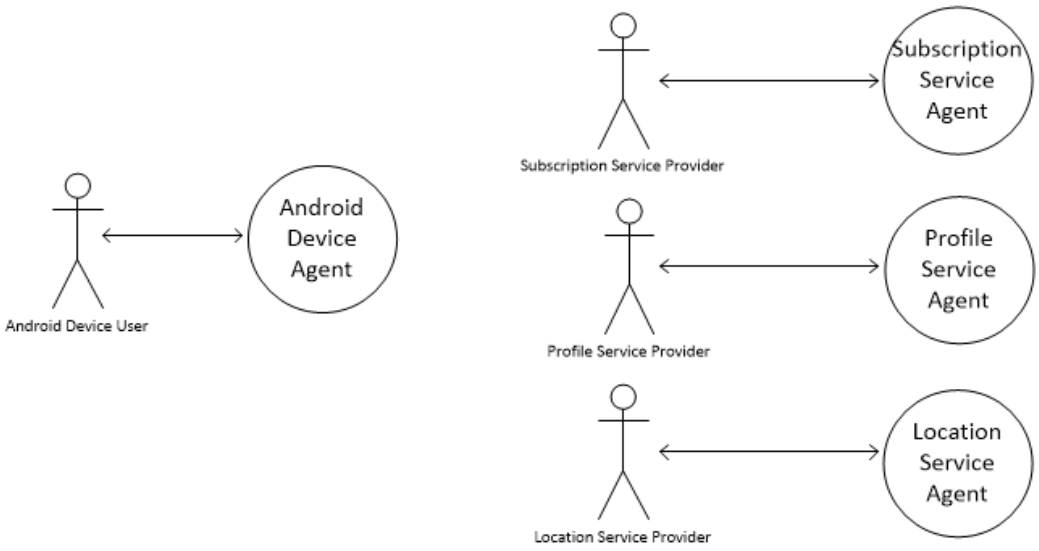
Figure 3.1 Use case diagram



3.2.3 Identifying agents

As recommended by the methodology [3] , all the users, devices and resources in use case diagram should be identified as agents. In an agent diagram, circle represents an agent, square means resources and use case actor is user. The first draft of agent diagram can be derived from use case diagram shown in Section 3.1.2:

Figure 3.2 Agent diagram



Although there are no legacy systems to support, the proposed system has to frequently interact with Android operating system. Among different approaches [35] to account for external systems in multi-agent system, the transducer approach is most suitable for proposed system on Android.

3.2.4 Identifying Responsibilities

Responsibilities of each agent are derived from use cases identified in Section 3.1.2 and listed in the Table 3.1.

Table 3.1 Responsibility table

Agent Type	***Responsibilities***
Subscription service agent	1. Register Android device and add it to the subscriber list. 2. Remove Android device from subscription list if it is not active for a period of time or connection lost. 3. Provide a list of active Android devices upon request. 4. Deregister Android device and remove it from the subscription list.
Location service agent	1. Maintain a location history for each Android device which optioned in location service. 2. Handle location updates from Android devices. 3. Facilitate profile service agent to answer the queries such as nearby Android devices.
Profile service agent	1. Maintain a profile database for all active Android devices. 2. Update Android device profile either periodically or by event. 3. Recommend optimized system settings to Android device based on its profile. 4. Send commands to start new agents on Android devices if certain conditions are met (For example, rooting is available or proactive measures need to be taken).
Android device agent	1. Register to the service provider. 2. Query nearby Android devices. 3. Update location to service provider. 4. Notify environment changes to service provider. 5. Switch to recommended system settings. 6. Deregister from the service provider.

3.2.5 Agent Refinement

The Android device agent can be further divided into four types of agents by applying the three agent refinement rules [3]: support, discovery, management and monitoring.

Profile agent:

This agent communicates with profile service provider directly. It maintains a local copy of profile information and receives commands from service provider. The agent will be active for the entire lifecycle of the agent platform on the Android device.

Sensor agent:

This agent detects environment changes such as network type, IP address, operating system (OS) version, or kernel version on the Android device and reports to the profile agent.

Location agent:

This agent subscribes to Android device GPS receiver events and sends location updates to location service provider.

Action agent:

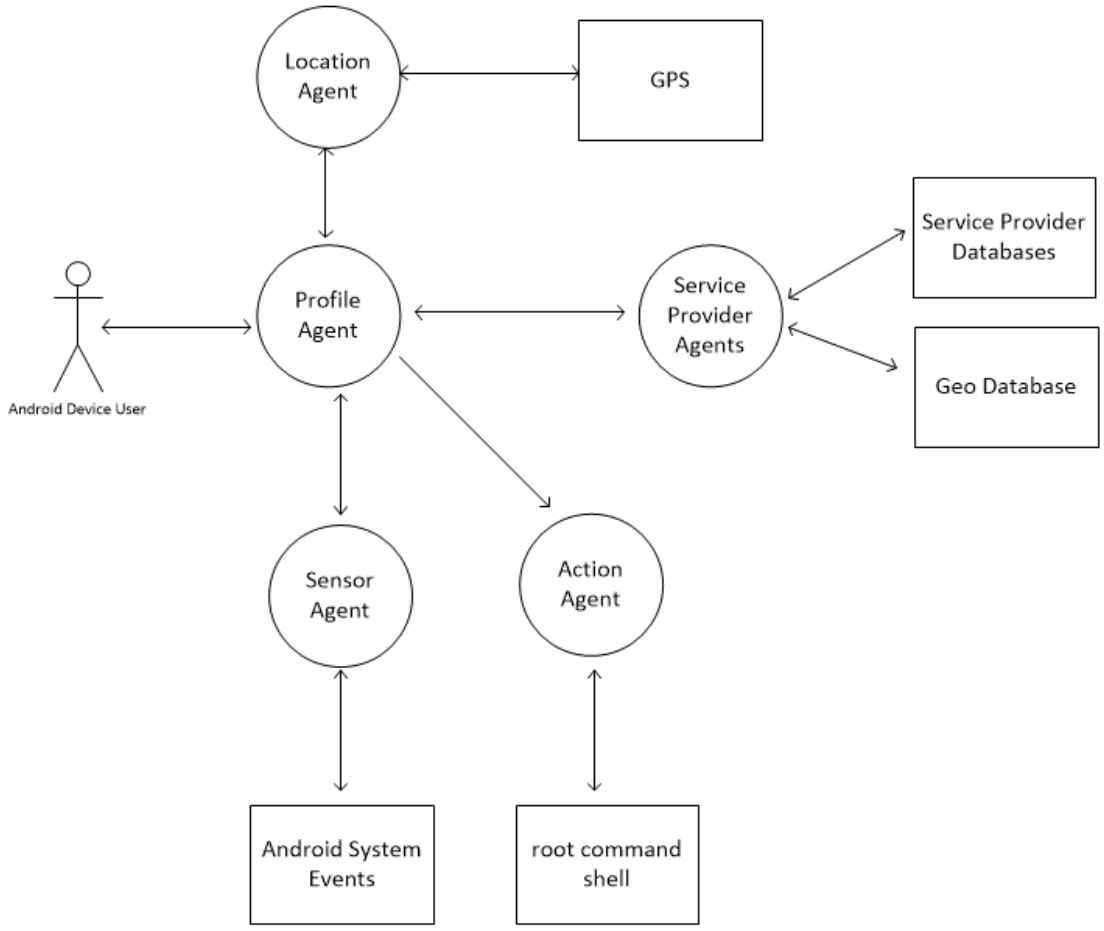
Depending on whether the Android device is rooted or not, the action agent can take various proactive measures upon the requests from profile agent. The proactive actions mentioned in Section 3.1.1 can all be performed by an action agent.

The refined responsibility table and agent diagram are listed as follows:

Table 3.2 Refined responsibility table

Agent Type	***Responsibilities***
Profile Agent	1. Register to subscription service provider. 2. Maintain a local copy of active android devices. 3. Report profile information to profile service provider. 4. Switch profile mode according to recommendations from profile service provider. 5. Keep tracking of resource level of all the agents started on the current Android device. 6. Respond to profile service provider commands. 7. Deregister from subscription service provider.
Sensor Agent	1. Check static settings such as root environment, Android version number. 2. Periodically detect network changes 3. Subscribe system sensor events. 4. Report changes to profile agent.
Location Agent	1. Subscribe local GPS services 2. Provide location updates to location service provider. 3. Query nearby android devices. 4. Query location history if necessary.
Action Agent	1. Snapshot current network statistics. 2. Capture network traffic. 3. Apply firewall rules. 4. Detect running processes/apps.

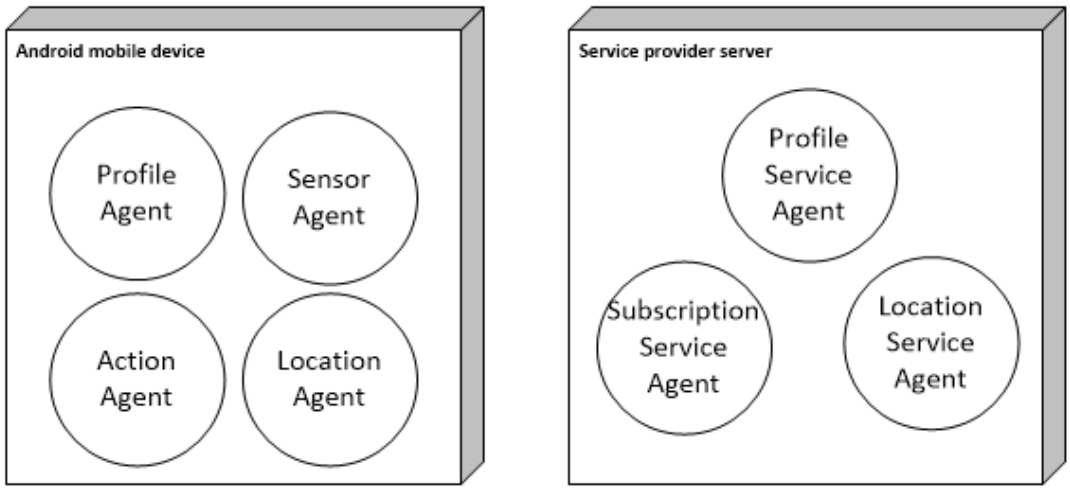
Figure 3.3 Refined agent diagram



3.2.6 Agent deployment diagram

The agents will be deployed to two different types of hosts: Android mobile devices and service provider servers. The agent deployment diagram is shown in Figure 3.4

Figure 3.4 Agent deployment diagram



3.3 System Design

3.3.1 Agents and interaction specification

Seven agents are identified at system analysis phrase and no more agent refinement is required at design phrase for the simplicity purpose. Specifying interactions between agents is the next step.

To define an agent interaction, following questions should be asked and answered:

1. Which agent's responsibility (Specified in the responsibility table) initiates the interaction?
2. What FIPA interaction protocol [28] fits the interaction best?
3. What are the roles of agents involved in the interaction? For example, initiator or responder?
4. Under what condition, the interaction will happen?

The interaction table below records some selected interaction specifications of a profile agent. All the responsibilities are defined in Table 3.2.

Table 3.3 Interaction table for profile agent

Interaction	*Responsibility*	*Interaction Protocol*	*Role*	*With*	*When*
Register	1	FIPA subscribe	I	Subscription service agent	Startup
Query resource level	5	FIPA Query	I	Sensor agent, action agent, location agents	Before starting agents
Retrieve active devices	2	FIPA request	R	Profile service agent	When new device subscribes
Switch profile mode	3	FIPA request	R	Profile service agent	Need to switch profile mode.

Similarly, the interaction tables for other agent types can be generated in a similar way. But we will not cover them here due to the space limitation.

3.3.2 Agent-Android resource interactions

Four types of agents are designed to run on Android devices: profile agent, sensor agent, action agent and location agent. The agents need to interact with resources on Android devices from time to time. These resources can be classified into two categories, namely static resources and run-time resources, depending on their status changing pattern. Table 3.4 depicts some important Android resources for the agents:

Table 3.4 Android resources

Static resources	*Run-time resources*
• SD card • Device ID • OS Version • Root command shell	• GPS receiver • Battery status • Network traffic and status • Network type

The interactions between agents and passive resources are simple and straightforward, thus the following two-level interaction design focuses on active Android resources only.

- Application level:
    - The application register intent filters to receive relevant system and application level broadcasting intents.
    - The application adds listeners to Android system services such as LocationManager.
    - Listeners will forward update information via application-level intents once system status updates are detected.
    - BroadcastReceiver in the application will process all the system events and updates from listeners then redistribute them to the agents via application level intent broadcasting.
- Agent level:
    - All the agents will keep a reference to application context.
    - Agents can periodic query updated information via application context, or
    - Register their own BroadcastReceiver and intent filters to retrieve updates from application-level broadcasting intents.

3.3.3 Agent-Android user interface interactions

All the human user interactions with proposed system are done via Android user interface. As a result, agents need constantly interact with android user interface to retrieve user inputs and send feedbacks to user. In fact, Android user interface can be treated as another active resource. Again, the intents, intent filters, broadcast receivers and listeners are the key elements in the interaction design. In addition, Android persistence storage such as preferences and files can also facilitate the interactions between the agents and Android user interface.

- Android user interface (usually, an activity in the context of Android framework) receive user inputs via various input events such as onClick(), onKey(), onTouch() and etc.
- Listeners registered in these input events will send application-level broadcasting intents if necessary.
- Agents with proper intent filters will receive user input updates.
- Agents can send update messages to the activity via broadcasting intents.
- With Broadcast Receivers and intent filter, the activity will present agents feedbacks to user.
- Activity write user's input to application shared preferences and agents read user's preferences back before any action.

Agents export output to a file on external storage such as SD card and activity will display the agents' output by reading certain folder on the SD card.

3.3.4 Agent behaviors

As recommended by the methodology, all the agent behaviors are extended from skeleton implementation classes provided by JADE such as OneShotBehaviour, CyclicBehaviour, TickerBehaviour and WakerBehaviour. In this section, we will on discuss some of the most important agent behaviours:

Devices manager behaviour extends from CyclicBehaviour and is one of the profile agent initial behaviours. This behaviour registers current Android device to subscription service agent and maintains a list of active android devices registered in the service provider. Register and deregister interactions between profile agent and subscription services agent are handled in this behaviour.

Update Android environment variables behaviour extends from TickerBehaviour and is the only behaviour of the sensor agent in a not rooted Android environment. This behaviour periodically updates android environment variables such as IP addresses and network type. The updates are sent to profile agent via broadcasting intents. Environment variable updates interaction between sensor agent and profile agent is handled by the behaviour.

Update agent location behaviour (extends OneShotBehaviour), *Receive server response behaviour* (extends SimpleBehaviour), *Wait server response behaviour* (extends ParallelBehaviour) are the three pillars of location agent behaviours. Together, these three behaviours allow location agent to update its current geography information to location service agent and wait for confirmation. The update agent location behaviour retrieves location information from application–level intent broadcasting and then sends an ACL message to location service agent. The wait server response behaviour is started after the ACL message is sent. In this ParallelBehaviour, two sub behaviours are added: receive server

response behaviour and a WakerBehaviour which will trigger a timeout event if there is no response from server after a certain period of time. There are two possible outcomes from this wait server response behaviour: server response is received or timeout event is triggered. In any cases, intent is generated and broadcasted to Android user interface.

The other agents' behaviours are designed similarly and not discussed in this thesis.

3.3.5 Ontology design and content language selection

Ontology, in the context of multi-agent systems, provides a domain model in which agents can communicate with each other [36]. In JADE, three interfaces are predefined for an application specific ontology to compose ACL messages: concept, predicate, and agent actions [37].

Classes implementing concept interface are equivalent to entity classes. In our system ontology, the following concepts are proposed for agent communication:

- Android device: includes all the Android environment variables needed by agents.

Slot Name	*Slot Type*	*Mandatory/Optional*
Is Rooted	Boolean	Mandatory
OS Version	String	Mandatory
Network Type	String	Optional
Resource Level	Integer	Mandatory
IP Address	String	Optional

- Android profile: defines current agent profile mode. For each profile mode, recommended settings are defined.

Slot Name	*Slot Type*	*Mandatory/Optional*
Profile Mode	Integer	Mandatory

- Location: the latitude and longitude that describe location information.

Slot Name	*Slot Type*	*Mandatory/Optional*
Latitude	Float	Mandatory
Longitude	Float	Mandatory

- Status: simple wrapper for string message.

Slot Name	*Slot Type*	*Mandatory/Optional*
Message	String	Mandatory

Predicate is a proposition to be verified. Here are the classes which implements predicate interfaces in the ontology:

- Registered: register the agent.

Slot Name	*Slot Type*	*Mandatory/Optional*
Agent	AID	Mandatory

- Deregistered: deregister the agent.

Slot Name	*Slot Type*	*Mandatory/Optional*
Agent	AID	Mandatory

Agent actions indicate some tasks needed to be performed by agents. Here are some agent actions classes in the ontology

- Update location: update the location of sender agent.

Slot Name	*Slot Type*	*Mandatory/Optional*
Updated location	Location	Mandatory

- Nearby devices: query nearby android devices by a given radius.

Slot Name	*Slot Type*	*Mandatory/Optional*
Current Location	Location	Mandatory

JADE includes support for two content languages: the SL language and the LEAP language [37]. The SL language is human readable and generally more compatible with different agent platforms. In contrast, the LEAP language codec is more compact due to its byte encoded nature. In the proposed system, the SL language is recommended because of its openness and ease of debugging.

Chapter 4: System Implementation

In this chapter we present our prototype implementation of the system. Starting from the preparation of Android and JADE agent development environment, we discuss the implementation details of Android user interface, ontology, agents on Android and agents on the server. Finally, in the last section, we integrate all the parts together into a multi-agent system and introduce some tools provided by JADE to test and debug the system.

4.1 Development environment

4.1.1 Android SDK

Android SDK (Software Development Kit) [38] provides all the necessary libraries and developer tools for developing Android applications. There are two download options available for all three major platforms (Windows, Mac OS X and Linux):

1. *ADT bundle*. This recently added option includes everything in a single package: The integrated development environment (IDE), namely Eclipse, with ADT (Android Development Tools) plugin; Android SDK tools for downloading and managing SDK packages; Android platform tools for debugging and testing. The latest SDK package and system image are also included.
2. *SDK tools only*. This is preferred option for the proposed system because it offers flexibility of choosing which SDK package to download and which IDE to use.

The key steps of preparing the Android SDK environment for developing the proposed system are listed as follows:

- Download the latest Eclipse and Android SDK tools.

- Install the following packages using Android SDK manager:
    - Android 4.0.3 (API 15) as targeting Android platform - SDK platform, ARM EABI v7a System Image, Intel x86 Atom System Image, Google APIs, Sources for Android SDK.
    - Android 2.3.3 (API 10) as backward compatibility testing platform – SDK Platform, Google APIs, Intel Atom x86 System Image.
    - Extras - Google USB driver, Intel x86 Emulator Accelerator (HAXM)
- Download and install Eclipse ADT plugin.
- Open Android virtual device manager in Eclipse and create two Android virtual devices (AVD): One AVD for ARM EABI v7a system image and anther one for Intel x86 Atom system image.

The Android SDK environment is now ready for prototype system development.

4.1.2 Android emulator

Android emulator is essential to Android application development because it allows developer to develop, debug and test without a physical Android device. However, the emulator needs to be optimized to meet the needs of developing the proposed system. Two important Android emulator tweaks are hardware acceleration and rooting Android emulator.

Hardware acceleration: Without hardware acceleration, the Android emulator runs painfully slow and is not responsive to user interactions. Android SDK tools rev 17 add the hardware acceleration support for Intel CPUs to solve the problem. Enabling hardware acceleration in Android emulator requires:

- Intel CPUs with Virtualization Technology (VT) support. (may also support AMD CPUs, but not focused in this thesis)
- Enable virtualization extensions in BIOS if it is not turned on by default.
- Use only x86 system image targeted AVD.
- Emulator cannot run inside a virtual machine (VM).
- Remove VM drivers installed by other system such as Virtualbox or VMWare.
- Enable GPU acceleration setting in AVD
- For Windows and Mac OS X, install the Intel hardware accelerated execution manager (HAXM) package in the extras. If the emulator shows message “HAX is working and emulator runs in fast virt mode”, then hardware acceleration is enabled.
- For Linux, install kernel-based virtual machine (KVM) package and start Android emulator from command line with following options: -qemu –enable-kvm

Rooting Android emulator: Both ARM and Intel Android 4.0.3 system images in Android SDK packages include su located at /system/xbin. To get a root shell on the emulator with Android debug bridge (ADB), the Android platform tools directory must be included in the PATH environment variable first. Assumes that an AVD named with arm_ics has been created, Android emulator can be started from command line:

```
dev$>emulator –avd arm_ics
```

to verify the emulator is up and ready to connect:

```
dev$>emulator –avd arm_ics
List of devices attached
emulator-5554   device
```

executes the following commands to enter the root shell on the emulator:

```
dev$>adb shell
#su
```

For the proposed system, agents need to access the root shell inside Android virtual machine which adb method cannot provide. To address this issue, Superuser [39] must be installed on the emulator. Superuser is an open source [40] software package which allows Android applications to have access to root shell. Two parts are included in the package, a customized su binary and a SuperUser Android application. To install Superuser on the emulator, download ARM or x86 Superuser package depends on one the emulator's system image setting and unzip it to a temporary folder. Then start emulator with larger partition size so su binary can be pushed to the emulator's system partition:

```
dev$>emulator –avd arm_ics –partition-size 200
```

remount the system partition, push su binary to system partition and grant execute permission to su binary.

```
dev$>adb remount
remount succeeded
dev$>adb push path\to\superuser\bin\su /system/xbin
114 KB/s (85096 bytes in 0.726s)
dev$>adb shell chmod 6755 /system/xbin/su
```

finally, install Superuser android application and rooting Android emulator is complete:

```
dev$>adb install path\to\superuser\app\Superuser.apk
145 KB/s (1500495 bytes in 10.099s)
        pkg: /data/local/tmp/Superuser.apk
Success
```

The two important Linux tools which are required for developing the proposed system are Busybox [41] and Tcpdump [42].

Busybox packs a collection of commonly used Linux utilities that Android doesn't come with by default in a simple executable. Optimized for size and limited resources, Busybox is ideal for embedded Linux environment like Android. Busybox utilities such as uname, netstat are important for agents to detect environment variables on rooted devices.

Tcpdump, works together with library libpcap, is a powerful command line tool for packet analysis. The Android SDK system images actually ship with Tcpdump binary. However, Tcpdump is still needed to be installed via adb for rooted Android devices. In the proposed system, agents can employ Tcpdump to capture network traffic and replay recorded pcap files.

The prebuild android binaries of both tools are available at OMAPpedia [43] [44], thus can be deployed on the emulator similar to su binary installation.

4.1.3 JADE development environment

Jade libraries and documentations are available and free for download (user registration required) from Tilab website [29]. The full package includes all the documents, samples, libraries and source code. The JADE development environment can be built by following the instructions provided by JADE administration guide [45]:

- Install latest Java Runtime Environment (JRE) or Java Development Kit (JDK)
- Set JAVA_HOME environment variable and add JRE or JDK directory to PATH environment variable.
- The jade.jar and commons-codec-1.3.jar libraries which come with JADE binary package must be added to CLASSPATH environment variable.

Having set these environment variables, executing the following commands will launch JADE agent platform:

```
dev$>java jade.Boot –gui
```

To develop JADE Android agent applications with Eclipse IDE, an ad-hoc version of JADE library, JadeAndroid.jar, is required [30]. The library can be downloaded from at Tilab

website and then referenced by the Android application project in Eclipse. The following Android application settings need to be adjusted:

- Add JadeAndroid.jar to java build path libraries property.
- In the Android manifest file, AndroidManifest.xml, declare MicroRuntimeService as service application component.

After these changes, the Android application can bind to Jade runtime service, create a split container and then start agents on Android devices.

4.2 System implementation

4.2.1 Android application

The most important user interface component in Android is activity. An activity uses the combinations of views and fragments to provide a screen with which the user can interact. The prototype consists of three activities: the main activity, the preference activity, and the file list activity.

Main activity:

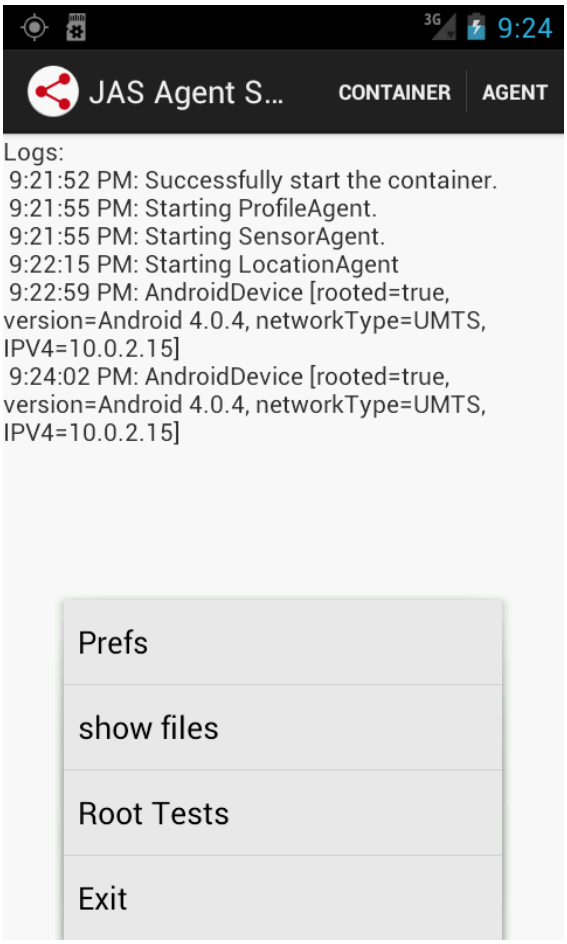
The main activity is defined with one relative layout resource (activity_main.xml) and one menu resource (mainmenu.xml). All the strings such as menu item names are declared in string resource (strings.xml).

In the relative layout, there is only one user interface component: a TextView named tv_log for displaying output information from agents. New messages are appended to the bottom of the TextView. The TextView will bring newest messages into the view by auto-scrolling if the messages exceed max lines defined. Here is the source code of TextviewLog helper method:

```
public void TextviewLog(String log) {
	final int scrollAmount = logTextView.getLayout().getLineTop(
			logTextView.getLineCount())
			- logTextView.getHeight();
	logTextView.append("\n " + JASHelpers.getCurrentTime() + ": " + log);
	if (scrollAmount > 0)
		logTextView.scrollTo(0, scrollAmount + 48);
	else
		logTextView.scrollTo(0, 0);
}
```

Menu items defined in mainmenu.xml resources are added to Action Bar as action items. The container action item starts JADE container on Android; Agent action item will create agents such as profile agent and sensor agent in the container; Preference action item brings preference activity allowing user to change the settings; Show files action item lists all temporary capture files on SD card; And exit action item will shut down the application.

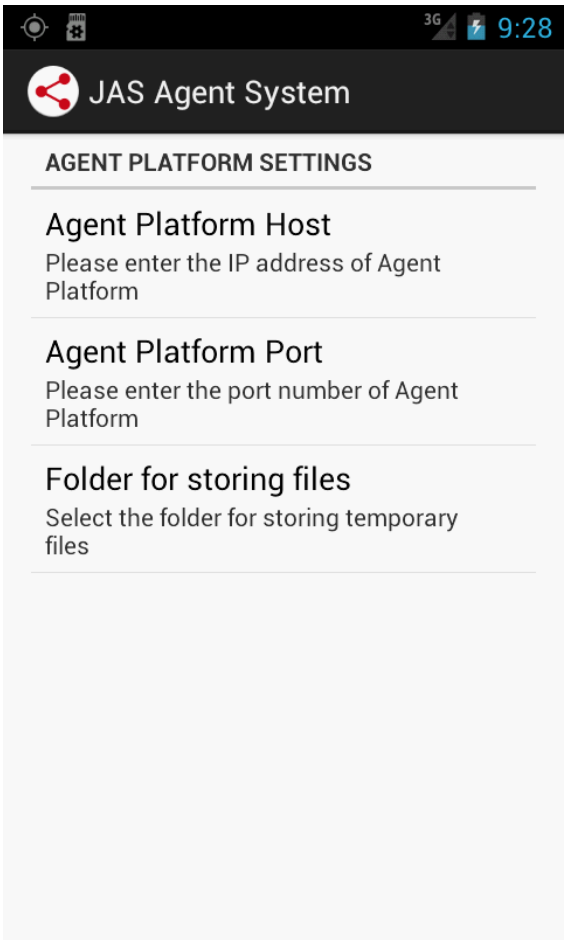
Figure 4.1 Android Application: Main Activity



Preference activity:

All the system settings can be adjusted via preference activity. As depicted in Figure 4.2, these settings include the IP address and port number of the agent platform and temporary folder for storing network capture files.

Figure 4.2 Android application: preference activity

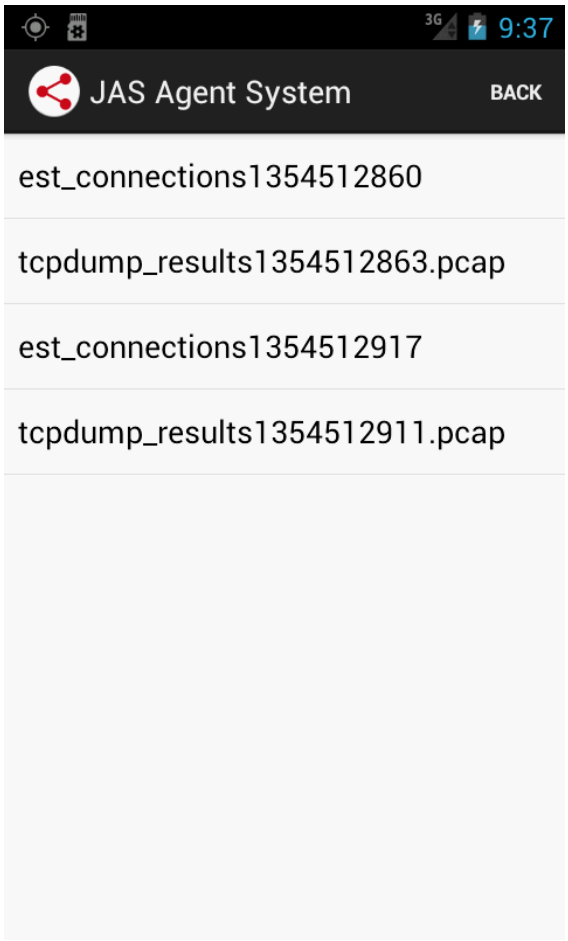


File list activity:

File list activity extends ListActivity class and references to a built-in XML list item layout simple_list_item_1. The activity reads temporary folder information from shared preferences,

then enumerates all the files in the folder and displays filenames in a list view. User can long hold on a file and upload it.

Figure 4.3 Android application: file list activity



The Android application requests permissions on behave of its agents in the manifest file. Figure 4.4 shows all the permission requested and Table 4.1 explains the rationale of these requests.

Figure 4.4 Android application: permission requested

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

Table 4.1 Agent – permission mapping table

Agent	*Permission*	*Rationale*
Profile agent, Location agent	Android.permission.INTERNET	Communicate with service provider agents
Location agent	Android.permission.FINE_LOCATION Android.permission.COARSE_LOCATION	Read GPS information
Sensor agent	Android.permission.ACCESS_NETWORK_STATE	Read network type

4.2.2 Ontology implementation

Protégé [46] is a suite of open source Ontology development tools. With its active developer community and many available plugins, Protégé is an ideal candidate for developing JADE application specific ontology. A tab widget plugin, OntologyBeanGenerator [47], can generate FIPA/JADE compliant ontology from domain models developed in Protégé. With the help of Protégé ontology editor and OntologyBeanGenerator plugin, ontology design in section 3.5.2 can be translated into a set of concept, predicate, agent actions and ontology classes ready to use in JADE environment.

JASOntology, the ontology definition class generated for the prototype system, extends JADE Ontology class in package jade.content.onto. The application of singleton pattern enforces there is only one instance of JASOntology in the system at any time. Constants are

defined in JASVocabulary interface and available to all ontology classes. The concept, predicate and agent actions classes implement corresponding interfaces respectively and register as schemas in the constructor of JASOntology. The ontology and content language codec must be initialized and registered to ContentManager before used by any agents:

```
slCodec = new SLCodec();
jasOntology = JASOntology.getInstance();
ContentManager contentManager = getContentManager();
contentManager.registerLanguage(slCodec);
contentManager.registerOntology(jasOntology);
contentManager.setValidationMode(false);
```

Then an agent can employ ontology registered to construct and send an ACL message:

```
ACLMessage message = new ACLMessage(performative);
message.setLanguage(slCodec.getName());
message.setOntology(jasOntology.getName());
getContentManager().fillContent(message,
            new Action(locationService, action));
message.addReceiver(locationService);
send(message);
```

4.2.3 Agents implementation

The following four types of agents are data collecting agents that are created on Android devices: profile agent, sensor agent, location agent and action agent. These agents interact with Android user interface and resources constantly thus they need to be implemented differently from other agents in the system. To achieve this, the application context is passed as a parameter when the agent is started from main activity on Android devices.

```
microRuntimeServiceBinder.startAgent(agentuid, agentname,
            new Object[] { this }, agentRuntimeCallback);
```

In the constructor, the agent then casts the first object in the parameter array to application context type and keeps a reference to the context in its whole lifecycle. With this reference, the agent can collaborate with Android user interface and resources at any time.

```
Object[] args = getArguments();
if (args != null && args.length > 0) {
      if (args[0] instanceof Context) {
            mainActivityContext = (Context) args[0];
            mainActivity = (MainActivity)mainActivityContext;
      }
}
```

The service provider agents are decision makers in the prototype system. They process information collected from agents running on Android devices and make recommendations depending on available resources. Three service provider agents are implemented: subscription service agent, profile service agent and location service agent.

The subscription service agent maintains a list of active Android devices registered in the agent platform. To achieve this, the agent needs to know when a new android device joined the agent platform. In JADE, the FIPA-subscribe-like interaction protocols are implemented by class SubcriptionInitiator and SubscriptionResponder in package jade.proto. By implementing the SubscriptionManager inner interface of SubscriptionResponder, the subscription service agent is notified when an Android device is registered or deregistered.

The profile service agent and location service agent are relatively simple since they don't need to deal with subscription requests. However, they must register themselves to the directory facilitator (DF) system agent so data collection agents can find them through a DF search. This works similarly to how service providers advertise themselves in the yellow page book in the real world. The agent DF registration is implemented as an OneShotBehavior:

```
ServiceDescription sd = new ServiceDescription();
sd.setType(JASOntology.SERVICE_AGENT_NAME);
sd.setName(getName());
sd.setOwnership("JASAgent");
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
dfd.addServices(sd);
DFAgentDescription[] dfds = DFService.search(myAgent, dfd);
if (dfds.length > 0) {
      DFService.deregister(myAgent, dfd);
}
DFService.register(myAgent, dfd);
```

Then an agent can perform a simple lookup in the DF and find the service provider it is looking for:

```
ServiceDescription sd = new ServiceDescription();
sd.setType(JASOntology.SERVICE_AGENT_NAME);
DFAgentDescription dfd = new DFAgentDescription();
dfd.addServices(sd);
DFAgentDescription[] dfds = DFService.search(this, dfd);
if (dfds.length > 0) {
      locationService = dfds[0].getName();
}
```

4.3 System integration and testing environment

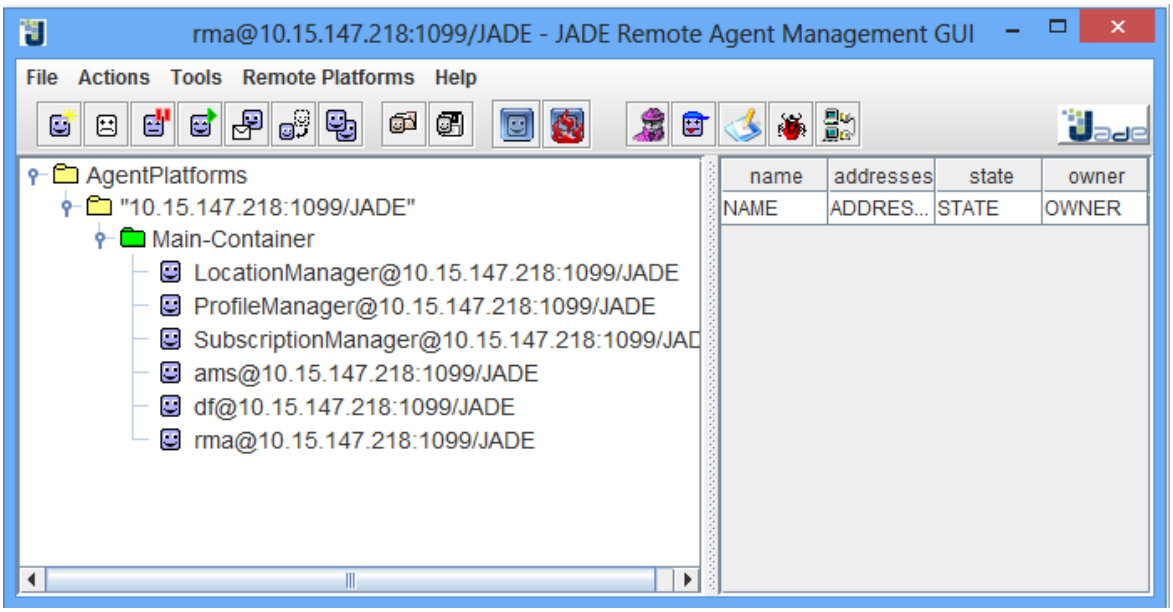
The prototype system naturally breaks down into two subsystems: A JADE agent platform with all the service provider agents runs on the servers and JADE runtime containers where all the data collection agents are created on Android devices.

A batch file (For Windows platform) or Unix script (For Linux or Mac OS X) is prepared for starting the JADE platform on the server.

```
java jade.Boot -gui –agents
“SubscriptionManager:com.jasagent.prototype.agents.services.Subscrip
tionServiceAgent;ProfileManager:com.jasagent.prototype.agents.servic
es.ProfileServiceAgent;LocationManager:com.jasagent.prototype.agents
.services.LocationServiceAgent”
```

Execute the batch or script file in the command line, a JADE agent platform will be started with all the service provider agents.

Figure 4.5 JADE agent platform with service provider agents



The JADE agent platform is now ready for data collection agent on Android devices to connect and interact. The Android device needs to know the IP address and port of the agent platform in order to connect. The default port number of JADE agent platform is 1099 and the IP address can be easily found since it is part of agent identifier (AID). These settings can be adjusted through the preferences screen of the Android application. In the main screen of the Android application, clicking the Container button on the action bar will create a container on the agent platform; clicking the Agent button will start data collection agents within the container.

Figure 4.6 JADE agent platform with agents from Android devices joined

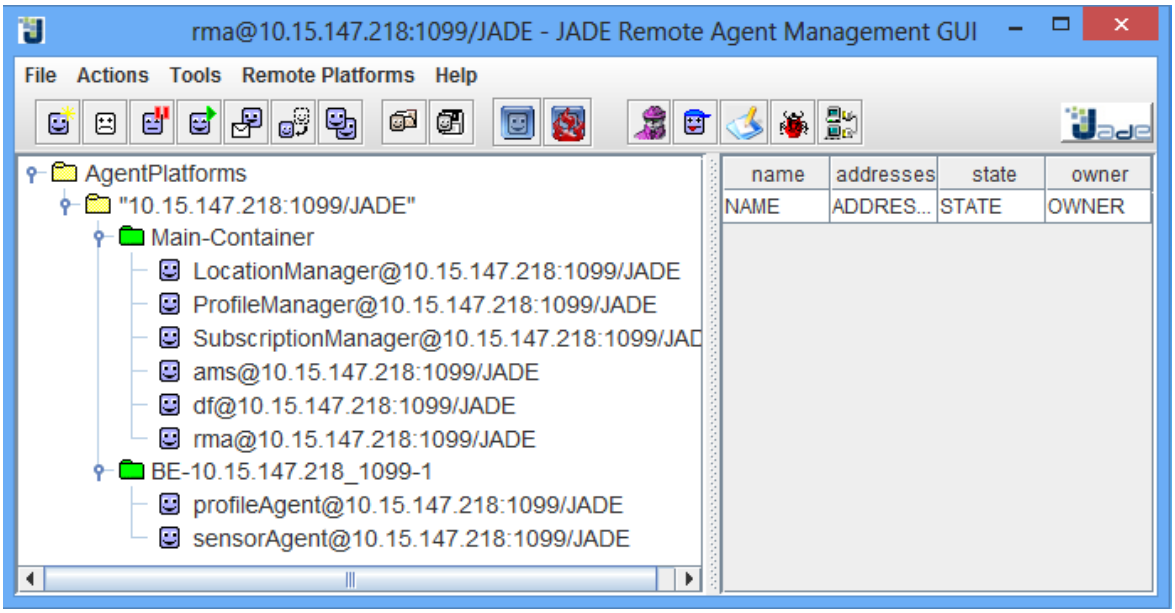
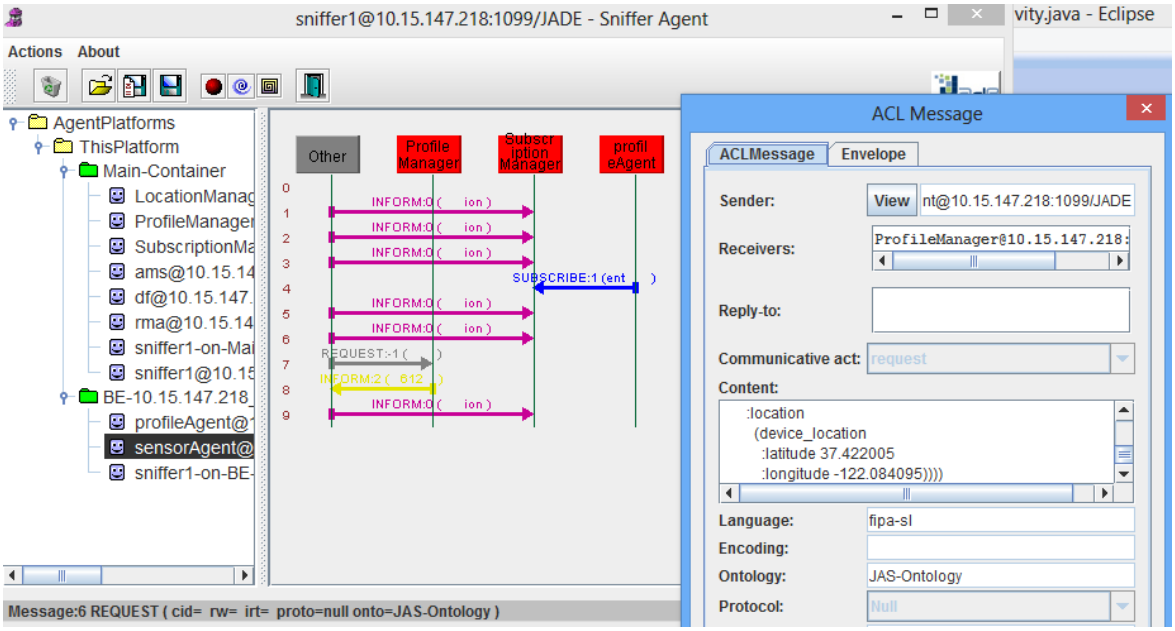


Figure 4.7 Sniffer agent showing an update location ACL message



The JADE remote agent management GUI offers some handy tools [45] for testing and debugging agent applications. For example, the dummy agent tool allows sending ACL

messages to any agent in the agent platform and stores them in a history list; the sniffer agent (as shown in Figure 4.7) can trace communications in a group of agents and show the details of ACL message exchanged. Together, these tools provide a powerful and convenient way to test and debug the prototype system.

Chapter 5: System Evaluation

In this chapter, we will evaluate the efficiency and performance of our prototype system through some experiments focused on battery consumption on Android devices. The key metrics examined in the experiments are processor usage, memory usage and network traffic.

5.1 Experiment setup

As depicted in the deployment diagram, the prototype consists of multiple components: An Android application component containing all the data collection agents, a server component that includes all the service provider agents and a network component (implied in the deployment diagram) that links two other components together. A common performance testing approach for SOA-based or component-based systems is end-to-end testing [48]. In our performance testing experiment, we follow end-to-end approach and cover all components involved with the following tools and devices.

5.1.1 Android application

The key performance metrics of the Android application are:

- CPU and memory usage
- Methods profiling
- Response time.

The application will be evaluated in the emulator first and then deployed on two rooted physical devices for further testing:

Table 5.1 Android testing devices

	Samsung Galaxy Tab 10.1	*Generic 7 inch Tablet*
Processor	Nvidia Tegra 2	Allwinner A10
Operating system	Android 4.0.3	Android 4.0.4
Memory	1Gb	512Mb
Storage	32Gb	4Gb

5.1.2 JADE agent platform server

JADE agent platform can be started with all service provider agents on any operating system using the batch file or Unix script provided with the prototype. Below, we indicate the details of the server used for our performance testing:

Table 5.2 JADE agent platform server configuration

Processor	Intel Core 2 Duo T9600
Memory	4Gb DDR3
Storage	128 Gb SSD
Operating System	Windows 8 pro 64bit
JADE platform	Jade 4.2.0
JDK	JDK 1.7.0_09

Performance testing was not done on the server as the released version of JADE has been thoroughly and independently tested, and its testing is beyond the scope of this thesis.

5.1.3 Android network emulator

The emulator control in Dalvik debug monitor server (DDMS) provides several simulation parameters such as speed and latency. By selecting different options, we can simulate a wide variety of network environments like GSM, GPRS, EDGE, HSDPA and etc. For the two

physical devices, we can only evaluate on the WIFI and 3G network. The key performance metrics of network are

- Packets sent
- Packets received
- Packets drops
- Average delay

These metrics can be evaluated using the network statistics tool in DDMS.

5.1.4 Android debug bridge (ADB)

Android debug bridge (ADB) [49] is a powerful tool which allows us to control and debug Android emulators or devices via command line. Here are some useful adb commands for our performance testing:

List attached emulators/devices - `adb devices`

Install Android application - `adb install \path\to\apk`

Show application's PID - `adb shell ps –A | grep <package name>`

Show CPU information for the process specified - `adb shell top`

Show process memory utilization - `adb shell procrank`

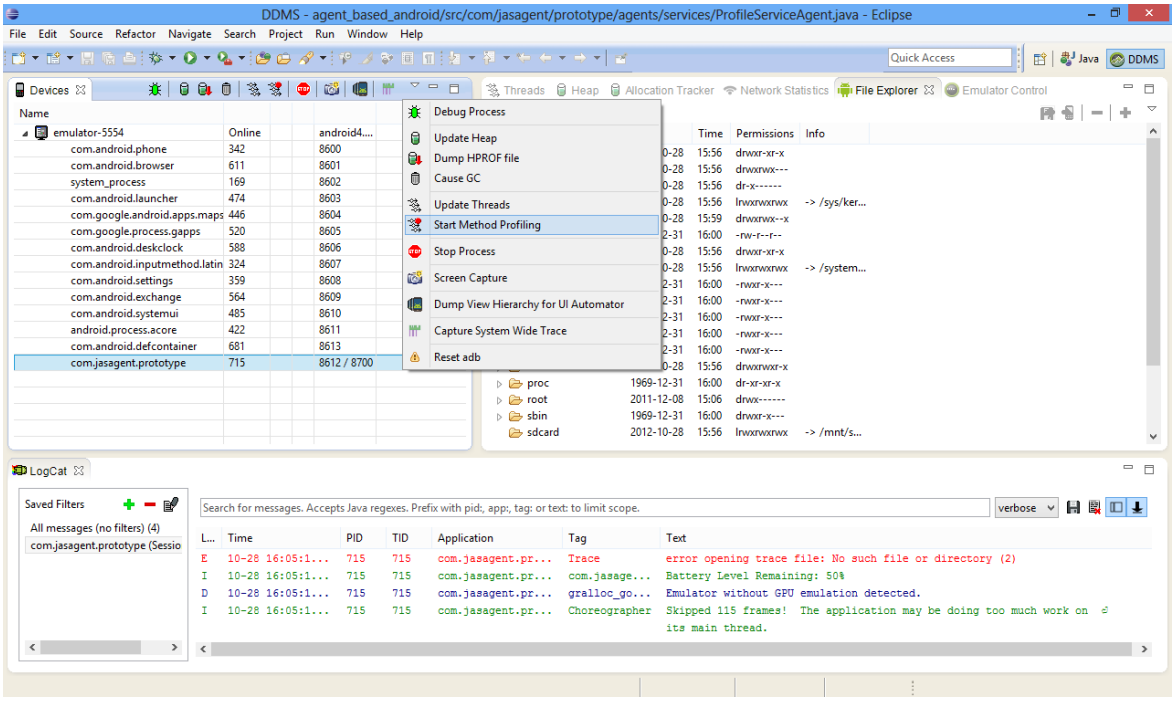
5.1.5 Dalvik debug monitor server (DDMS)

Dalvik debug monitor server (DDMS) [50] is a debugging tool shipped with the Android SDK. The tool works with both the emulator and physical devices which have USB debugging option turned on. It provides many features and capabilities, and here we list some features that are important for our performance testing:

- Screen capture on the device
- Thread, heap and process information
- Network environment simulation
- Logcat integration
- Location data spoofing
- File explorer to manage files on the device
- Method profiling which requires Android 2.2 or later
- Network traffic tool which requires Android 4.0 or later

DDMS can be started either from Eclipse or via command line. Figure 5.1 shows the DDMS perspective in Eclipse.

Figure 5.1 Dalvik debug monitor server



5.2 Experiment methodology

To evaluate the power consumption of our prototype, we designed the following performance test focusing on three key performance metrics of the Android application: CPU usage, memory usage, and network traffic. In this section, we explain how to measure these metrics with the tools available and list the steps of our experiment.

The location agent in our prototype subscribes to Android GPS events and reports its location to the location service agent in real time. This is one of the most CPU and network intensive tasks in our system. To evaluate the performance of this feature, a sample set of location data with timestamps is required. DDMS supports simulation location data by sending mock locations to the emulator in three geo-location data types: manually specifying longitude and latitude values, GPS eXchange file (GPX), and Keyhole Markup Language file (KML). However, the location control tool in DDMS is not compatible with physical Android devices. To address this, we developed a helper Android application generating mock location events every one to five seconds using a time-stamped location dataset. Our prototype system was subsequently able to support mock locations.

Linux system command tool *top* is ported to Android and shipped with all Android images. Entering top in command line will list all the running processes in the system sorted by their CPU usage. We use the following adb command to periodically monitor the CPU usage of the prototype system:

```
adb shell top –d 1 –n 1 | grep com.jasagent >> cpu_usage.log
```

Another Android command line tool *procrank* is essential to our experiment. The *procrank* [51] command shows a summary of process memory utilization including virtual memory set size (Vss), resident memory set size (Rss), proportional memory set size (Pss) and unique

memory set size (Uss), sorted by Vss. The two important memory parameters for an Android application process are Pss and Uss (Which the top command doesn't provide). The following adb shell command is used for evaluating memory usage of the prototype system:

```
adb shell procrank | grep com.jasagent >> memory_usage.log
```

The network statistics tool that comes with DDMS provides detail network usage of any Android application over time. The tool tracks when the Android application makes network requests and presents results in a real-time graphic user interface. We use this tool to analyze the data transfer pattern of the prototype system.

We also prepare a bash script to monitor the CPU and memory usage of the prototype system every second by executing *top* and *procrank* commands and appending the output to log files. Here is the procedure of a single performance test:

- Start the JADE agent platform on the server with all three service agents; write down the server IP address.
- Start the Android prototype on the Android device and provide the server IP address in the preferences screen.
- Execute the bash script we prepared to record CPU and memory usage.
- In the DDMS, start capture network traffic using the network statistics tool.
- In the Android prototype, connect to the JADE agent platform and create data collection agents.
- Generate mock GPS location events using the helper Android application we developed.
- Stop the test and collect all the data results

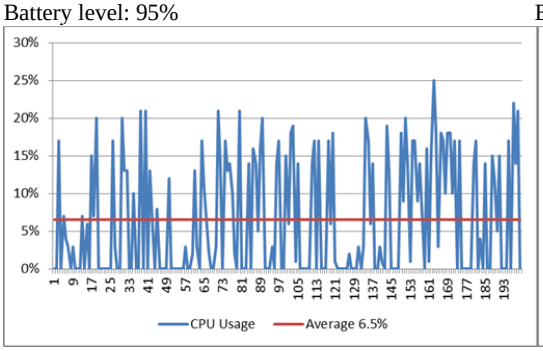
The performance test is repeated for two different battery levels, 95% and 40%, to evaluate the prototype system performance under high and low battery situations, and two physical Android devices for comparisons.

5.3 Experiment results and analysis

Following the procedure designed in the previous section, we evaluated the prototype system on a generic 7-inch Android tablet and produced diagrams showing results in the three key metrics we focused on: CPU usage (Figure 5.2), memory usage (Figure 5.3), and network traffic pattern (Figure 5.4). The same performance test is repeated on another Samsung galaxy tablet 10.1 device and we observed similar CPU/memory and network usage trends.

Figure 5.2 CPU usage

Battery level: 95%



Battery level: 40%

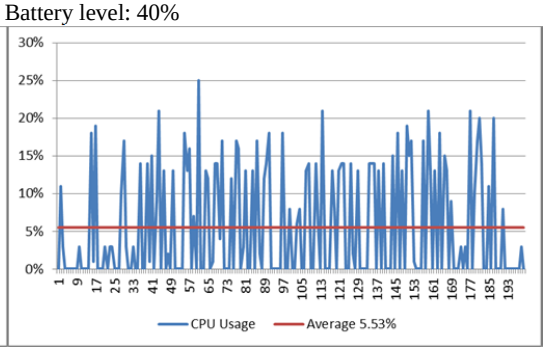
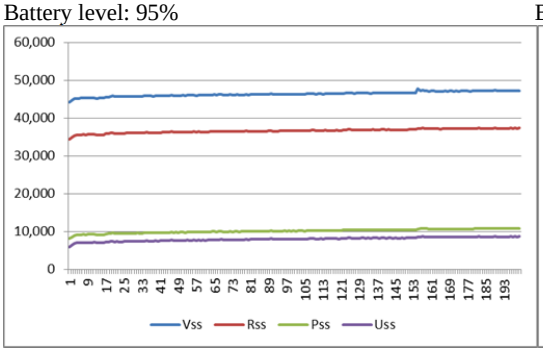


Figure 5.3 Memory usage

Battery level: 95%



Battery level: 40%

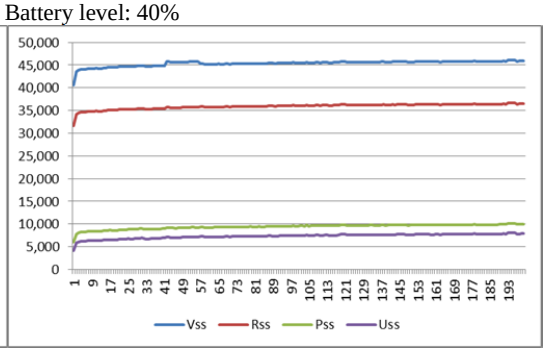
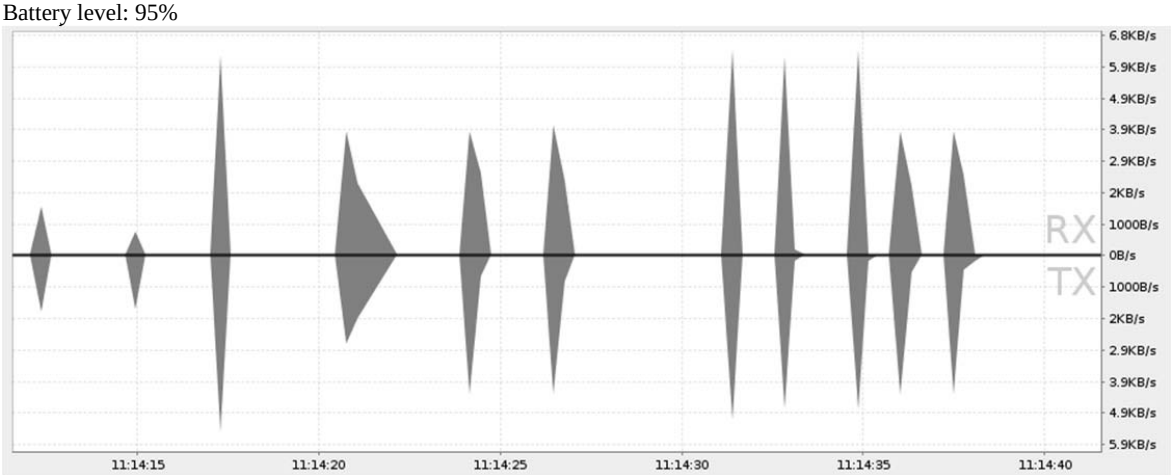
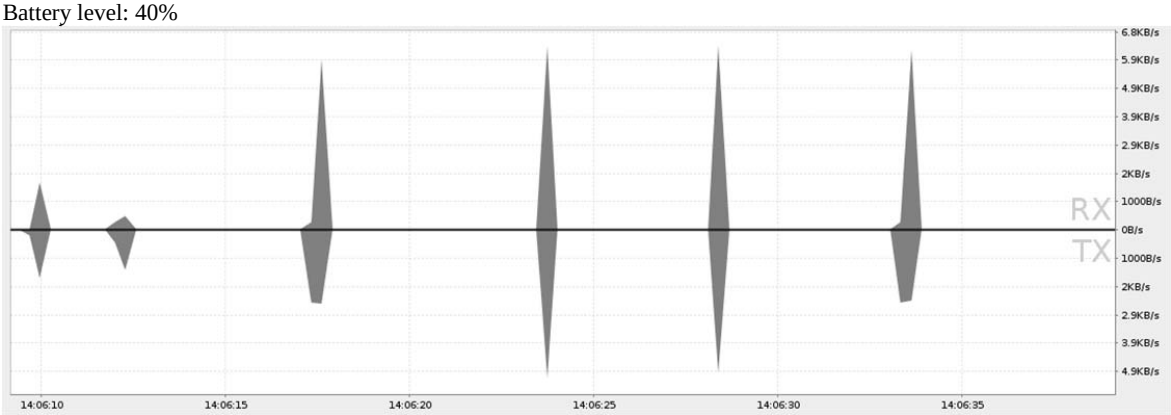


Figure 5.4 Network traffic pattern

Battery level: 95%



Battery level: 40%



From these diagrams, we can observe that at the high battery level (95%), the prototype system tends to consume more CPU power, more memory and more frequently communications with service provider agents. For example, the average CPU usage at high battery level is 6.5% comparing to 5.53% average at low battery level; the network communication frequency drops to almost half at 45% battery level. The profile agent in the prototype system closely monitors the battery level of the Android device and adjusts the behaviours of data collection agents accordingly. At the low battery level, the profile agent

will reduce the frequency of environment detection tasks performed by sensor agents, avoid creation of resource intensive action agents, combines profile and location update information into one ACL message, etc. These measures are proven to be effective by our performance test results. In summary, the prototype system dynamically adjusts its behaviours accordingly the available system resources, thus reducing CPU usage, memory usage and network communication at lower battery level which will lead to longer battery life in general.

Chapter 6: Conclusions and Future Work

6.1 Conclusions

Smartphones and tablets are becoming more and more important in our daily lives. With their ubiquitous presence in both the workplace and our everyday lives and increasing processor power, the mobile security threats we will face in the near future will not be simply collecting privacy information or sending premium SMS messages, but large scale denial-of-service attacks, identify theft or even worse. As the mobile operating system with the largest market share, Android has its own unique security requirements and challenges which traditional computer security systems cannot address.

In this thesis, we proposed a multi-agent security system built on JADE framework specially targeting resource constrained Android mobile devices. Agents in the system are fully aware of resources available and intelligently adjust their computing needs accordingly. Legacy security systems can be easily extended as service provider agents if their data collection components can be extracted and distributed.

The viability of the system is proven in the implementation details of a prototype following Android development guidelines and a methodology tailored for the JADE development framework. The prototype consists of an Android application and a batch file or Unix script to start the JADE agent platform with three types of service provider agents. In the rooted android environment, we also experimented in capturing real-time network status via Busybox, Tcpdump and RootTools [52] library.

To evaluate our system, we added randomly generated resource levels to each data collection agents on Android except the profile agent. The profile agent then monitors the

Android battery level constantly and makes intelligent decisions on how many and how often sensor agents, location agents and action agents need to be created according to their resource level requirement. The profile service agent can also give recommendations based on information provided by the profile agent and predefined rules. Early experiment results are promising: the prototype can reduce processor and memory usages when battery level is low while still maintaining minimum data collection requirements for the security needs.

6.2 Future work

Several interesting areas can be investigated in the future to extend our system. The first emerging technology that deserves close investigation is Bluetooth Low Energy (BLE) [53]. BLE is part of Bluetooth 4.0 standard which aims at providing ultralow power communications mechanisms to energy restricted devices within a short range. Some BLE-enabled sensors are able to communication using a coin cell battery even up to two years. "Internet of Things" [54] attempts to extend the scope of internet to every physical object in the world. IPv6 specification [55] solves the "unique addressability of things" problem and BLE or similar technologies will offer a low energy level communication solution among the physical objects with sensors. The advancement in this area creates many new possibilities: point-to-point interaction between smart BLE objects with smart phones. Currently, BLE hasn't been supported by the Android operating system yet, but its potential applications in the proposed multi-agent platform should not be overlooked. Imagine in a world full of BLE enabled sensors. Data collecting agents in our system can interact with sensors around the mobile device in addition to sensors on the device. This will create many more interesting application scenarios.

Porting more traditional Linux security solutions as service provider agents is another interesting area of research. In this thesis, we only presented a limited number of service provider agents in the prototype to verify the system. By separating data collection from the security analysis process, our system can easily be extended to support many other security systems such as cloud antivirus, intrusion detection system, botnet detection, the Honeynet project, etc.

Finally, agent mobility is another promising area of exploration. Due to the limited resources we have, agent mobility is not implemented in our system although the JADE framework has full support of it. If an agent, especially data collecting agents, can move freely from one Android device to another, some interesting scenarios can be defined and explored. For example, a special sensor agent can traverse through all the nearby Android devices and try to detect whether a suspicious application is installed in all the devices. If this is confirmed, another agent can be dispatched and sent to these potentially compromised devices to do some further investigations. By allowing agents moving freely among devices, the system can be extended to tackle more complicated security issues such as Botnet.

Bibliography

[1] FIPA, "FIPA - Standard Status Specifications," 2002. [Online]. Available: http://www.fipa.org./repository/standardspecs.html. [Accessed 1 december 2012].

[2] F. Bellifemine, A. Poggi and G. Rimassa, "JADE–A FIPA-compliant agent framework," *Proceedings of PAAM*, vol. 99, no. 97-108, p. 33, 1999.

[3] M. Nikraz, G. Caire and P. A. Bahri, "A methodology for the development of multi-agent systems using the JADE platform," *International Journal of Computer Systems Science & Engineering*, vol. 21, no. 2, pp. 99-116, 2006.

[4] OHA, "Open Handset Alliance Releases Android SDK," 12 November 2007. [Online]. Available: http://www.openhandsetalliance.com/press_111207.html. [Accessed 1 December 2012].

[5] IDC, "Android Marks Fourth Anniversary Since Launch with 75.0% Market Share in Third Quarter, According to IDC," 1 November 2012. [Online]. Available: http://www.idc.com/getdoc.jsp?containerId=prUS23771812#.UMKYdIPO39Y. [Accessed 1 December 2012].

[6] OHA, "Android Open Source Project," Android Open Source Project, 11 November 2007. [Online]. Available: http://source.android.com/. [Accessed 1 december 2012].

[7] E. Fu, "Google removes 27 apps from Android Market in response to RuFraud malware threat," 14 December 2011. [Online]. Available: http://www.theverge.com/2011/12/14/2635274/google-malware-apps-android-market-rufraud. [Accessed 1 December 2012].

[8] D. Rosenberg, "CarrierIQ: The Real Story," 05 12 2011. [Online]. Available: http://vulnfactory.org/blog/2011/12/05/carrieriq-the-real-story/. [Accessed 01 12 2012].

[9] "New Virus SMSZombie.A Discovered by TrustGo Security Labs," TrustGo Security Labs, 15 August 2012. [Online]. Available: http://blog.trustgo.com/SMSZombie/. [Accessed 1 December 2012].

[10] AV-TEST Institute, "Test Report: Anti-Malware solutions for Android," 15 March 2012. [Online]. Available: http://www.av-test.org/fileadmin/pdf/avtest_2012-02_android_anti-malware_report_english.pdf. [Accessed 1 December 2012].

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pp. 1-6, 2010.

[12] D. Schreckling, J. Posegga, J. Köstler and M. Schaff, "Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android," *Information Security Theory and Practice. Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems*, pp. 208-223, 2012.

[13] M. Nauman, S. Khan and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pp. 328-332, 2010.

[14] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev and C. Glezer, "Google android: A comprehensive security assessment," *Security & Privacy, IEEE*, vol. 8, no. 2,

pp. 35-44, 2010.

[15] "SELinux," February 2012. [Online]. Available: http://selinuxproject.org/page/Main_Page. [Accessed 1 December 2012].

[16] A. Shabtai, Y. Fledel and Y. Elovici, "Securing Android-powered mobile devices using SELinux," *Security & Privacy, IEEE,* vol. 8, no. 3, pp. 36-44, 2010.

[17] A. D. Schmidt, H. G. Schmidt, J. Clausen, K. A. Yuksel, O. Kiraz, A. Camtepe and S. Albayrak, "Enhancing security of linux-based android devices," *in Proceedings of 15th International Linux Kongress. Lehmann,* 2008.

[18] "Application Fundamentals | Android Developers," [Online]. Available: http://developer.android.com/guide/components/fundamentals.html. [Accessed 1 December 2012].

[19] W. Enck, M. Ongtang and P. McDaniel, "Understanding Android Security," *Security & Privacy, IEEE,* pp. 50-57, 2009.

[20] R. Cannings, "An Update on Android Market Security," 5 March 2011. [Online]. Available: http://googlemobile.blogspot.ca/2011/03/update-on-android-market-security.html. [Accessed 1 December 2012].

[21] T. Vidas, N. Christin and L. Cranor, "Curbing android permission creep," *Proceedings of the Web,* vol. 2, 2011.

[22] J. Larimer and K. Root, "Google I/O 2012 - Security and Privacy in Android Apps - YouTube," 29 June 2012. [Online]. Available: http://www.youtube.com/watch?v=RPJENzweI-A. [Accessed 1 December 2012].

[23] M. Wooldridge, An introduction to multiagent systems, Wiley, 2002.

[24] M. Wood and S. DeLoach, "An overview of the multiagent systems engineering methodology," *Agent-Oriented Software Engineering*, pp. 1-53, 2001.

[25] "the Foundation for Intelligent Physical Agents," FIPA, 8 June 2005. [Online]. Available: http://www.fipa.org/. [Accessed 1 December 2012].

[26] FIPA, "FIPA Agent Management Specification," 18 March 2004. [Online]. Available: http://www.fipa.org/specs/fipa00023/SC00023K.html#_Toc75950978. [Accessed 1 December 2012].

[27] "FIPA ACL Message Structure Specification," 12 December 2002. [Online]. Available: http://www.fipa.org/specs/fipa00061/SC00061G.html. [Accessed 1 December 2012].

[28] FIPA, "FIPA Interaction Protocol Specifications," 2002. [Online]. Available: http://www.fipa.org/repository/ips.php3. [Accessed 1 December 2012].

[29] "Jade - Java Agent DEvelopment Framework," Tilab, [Online]. Available: http://jade.tilab.com/. [Accessed 16 November 2012].

[30] G. Caire, G. Iavarone, M. Izzo and K. Heffner, "JADE Tutorial: JADE Programming for Android," 14 June 2012. [Online]. Available: http://jade.tilab.com/doc/tutorials/JadeAndroid-Programming-Tutorial.pdf. [Accessed 16 Novemeber 2012].

[31] J. Odell, "Objects and agents compared," *Journal of object technology*, vol. 1, no. 1, pp. 41-53, 2002.

[32] M. Wooldridge, N. R. Jennings and D. Kinny, "The Gaia methodology for agent-oriented analysis and design," *Autonomous Agents and Multi-Agent Systems*, vol. 3, no. 3, pp. 285-312, 2000/9/1.

[33] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia and J. Mylopoulos, "Tropos: An agent-oriented software development methodology," *Autonomous Agents and Multi-Agent Systems,* vol. 8, no. 3, pp. 203-236, 2004/5/1.

[34] S. Loach and M. Wood, "Developing multiagent systems with agentTool," *Intelligent Agents VII Agent Theories Architectures and Languages,* pp. 30-41.

[35] M. R. Genesereth and K. S. P., "Software agents," *Commun. ACM,* vol. 37, no. 7, pp. 48-53, 147, 1994.

[36] M. N. Huhns and M. P. Singh, "Ontologies for agents," *Internet Computing, IEEE,* vol. 1, no. 6, pp. 81-83, 1997/11.

[37] G. Caire and D. Cabanillas, "JADE Tutorial Application-defined Content Languages and Ontologies," 15 April 2010. [Online]. Available: http://jade.tilab.com/doc/tutorials/CLOntoSupport.pdf. [Accessed 1 December 2012].

[38] "Android SDK," Google, [Online]. Available: http://developer.android.com/sdk/index.html. [Accessed 1 December 2012].

[39] "Superuser," AndroidSU.com, 06 March 2011. [Online]. Available: http://androidsu.com/superuser/. [Accessed 1 December 2012].

[40] ChainsDD, "ChainsDD/Superuser · GitHub," github.com, [Online]. Available: https://github.com/ChainsDD/Superuser. [Accessed 1 December 2012].

[41] D. Vlasenko, "BusyBox," [Online]. Available: http://www.busybox.net/. [Accessed 1 December 2012].

[42] "TCPDUMP/LIBPCAP public repository," [Online]. Available: http://www.tcpdump.org/. [Accessed 1 December 2012].

[43] "Android Installing Busybox Command Line Tools," omappedia.org, 17 April 2012. [Online]. Available: http://omappedia.org/wiki/Android_Installing_Busybox_Command_Line_Tools. [Accessed 1 December 2012].

[44] "USB Sniffing with tcpdump," omappedia.org, 25 September 2012. [Online]. Available: http://omappedia.org/wiki/USB_Sniffing_with_tcpdump. [Accessed 1 December 2012].

[45] F. Bellifemine, G. Caire, T. Trucco, G. Rimassa and R. Mungenast, "JADE Administator's Guide," 8 April 2010. [Online]. Available: http://jade.tilab.com/doc/administratorsguide.pdf. [Accessed 16 November 2012].

[46] "The Protégé Ontology Editor and Knowledge Acquisition System," Stanford Center for Biomedical Informatics Research , 12 Janurary 2012. [Online]. Available: http://protege.stanford.edu/. [Accessed 1 December 2012].

[47] C. Van Aart, "ProtegeWiki: Ontology Bean Generator," 27 February 2009. [Online]. Available: http://protege.cim3.net/cgi-bin/wiki.pl?OntologyBeanGenerator. [Accessed 1 December 2012].

[48] Y. Sikri, "End-to-End Testing for SOA-Based Systems," February 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc194885.aspx. [Accessed 1 December 2012].

[49] "Android Debug Bridge," [Online]. Available: http://developer.android.com/tools/help/adb.html. [Accessed 1 December 2012].

[50] "Using DDMS," [Online]. Available: http://developer.android.com/tools/debugging/ddms.html. [Accessed 1 December 2012].

[51] "Android Memory Usage - eLinux.org," 15 September 2011 . [Online]. Available: http://elinux.org/Android_Memory_Usage. [Accessed 1 December 2012].

[52] "roottools," 12 September 2012. [Online]. Available: http://code.google.com/p/roottools/. [Accessed 1 December 2012].

[53] "Low Energy | Bluetooth Technology Website," [Online]. Available: http://www.bluetooth.com/Pages/low-energy.aspx. [Accessed 1 December 2012].

[54] K. Ashton, "That ‘Internet of Things’ Thing," *RFiD Journal,* vol. 22, pp. 97-114, 2009.

[55] S. E. Deering, "Internet protocol, version 6 (IPv6) specification," 1998. [Online]. Available: http://tools.ietf.org/html/rfc2460. [Accessed 1 December 2012].