Project Number: IST-1999-20147
Project Title: CRUMPET, Creation of User Friendly Mobile Services Personalised for Tourism
Deliverable Type: P

CEC Deliverable Number: IST-1999-20147/UHe/WP2/D21
Contractual Date of Delivery to CEC: 31/12/2000
Actual Date of Delivery to CEC: 
Title of Deliverable: Design of Small Footprint Agent Platform
WP Contributing to Deliverable: WP2
Nature of Deliverable: Report
Authors: Jani Leinonen (UHe), Kimmo Raatikainen (UHe), Sasu Tarkoma (UHe), Milla Mäkeläinen

Abstract: This deliverable concentrates on the design of the CRUMPET agent platform for small devices, which supports the use of CRUMPET services. The end-users of the CRUMPET system will use the platform to interact with various tourism services on the network. The CRUMPET platform, or MicroFIPA-OS, is based on the open source Java-based FIPA-OS agent platform. This deliverable examines the Java solution to be used in constrained environments, the Linux operating system on small PDA devices and presents a conclusion on these technologies and their role in the CRUMPET small footprint agent services. This document examines and discusses different design alternatives for the MicroFIPA-OS, including optimization of the platform components and their distribution using different partitioning mechanisms. The role of profiles is examined and a distribution mechanism that relies on profiles is presented.

Keyword List: multi agents, software agents, FIPA, agent platform
Executive Summary

The CRUMPET project uses software agent technology in order to create and deploy various tourism-based services for a mobile audience. This deliverable concentrates on the design of the CRUMPET agent platform for small devices, which supports the use of CRUMPET services. The design goal of this document is to have a distributed agent platform that scales to a large number of nomadic users and supports the use of CRUMPET services on small devices. As the mobile devices are limited in processing power, memory and persistent storage we need to place agent platform functionality on systems on the fixed network to some extent.

The CRUMPET agent platform is based on the open source FIPA-OS software. Since FIPA-OS uses the Java programming language, Java is also the choice for the CRUMPET platform, also called MicroFIPA-OS. However, since the CRUMPET end users will use small and resource constrained devices the existing system needs to be optimized and modified.

The operating system, device, execution environment and device extensibility all affect the design of the platform and the CRUMPET services. The target device of the CRUMPET platform is the Compaq iPAQ, which represents the current high-end in PDA devices with the 206 MHz StrongARM processor and TFT display. One of the key issues with iPAQ is extensibility: the device supports PCMCIA cards and CompactFlash cards through a clip-on expansion-pack architecture.

The CRUMPET platform will be built on the open source Linux operating system, which is available for the Compaq iPAQ (and a variety of other small devices). This deliverable examines several Java platforms and environments for small devices. The CRUMPET platform will be based on PersonalJava. The benefits of the PersonalJava platform are portability and it provides approximately the standard Java JDK 1.1 functionality.

In order to have a scalable, interoperable and adaptable software platform for the CRUMPET agents that interact with the tourism services on the network, an agent platform that relies on profile based information is needed. Profiles provide the necessary configuration information, for example where different components are located, what kinds of platform specific components are available, etc.

The design of the MicroFIPA-OS consists of several improvements and optimizations to the current FIPA-OS platform that is based on Java 1.2 and intended to be used in a traditional computing environment. The current FIPA-OS code is simplified by introducing lightweight components that may be used instead of the original components with the help of the profile based configuration information. Lightweight plug-ins include transport protocol implementations, parsers, task and conversation management, conversations, tasks. These components may be used in future FIPA-OS releases. The available components are defined in agent platform profiles. Hardware profiles define the system performance, and output and input capabilities. Agent platform profiles define the current limitations on the number of tasks, what kind of transport components should be used, what interaction protocols may be used etc.

It is envisaged that the functionality of the MicroFIPA-OS platform may be split between the mobile node and the fixed network. We have considered the two architectural issues: the fully functional platform that is capable of stand-alone operation and the partial platform that needs the Access Node. The development strategy of MicroFIPA-OS is seen as a continuous process of analysis and implementation, where the device performance issues and demands affect the final configuration of the CRUMPET system. This deliverable imposes no set boundary, but rather allows the implementation of a scalable architecture that can support partitioning to a varying degree depending on the applications and how much memory and processing power is needed for other tasks.
GLOSSARY

ACC
Agent Communication Channel manages inter-platform communication and is specified by FIPA as one of the mandatory parts of an agent platform.

ACL
Agent Communication Language that is used between agents.

Access Node
The CRUMPET Access Node serves as the portal and interface to all tourism services developed in the CRUMPET project.

Agent shell
Agent shell is the basic building block of agents in FIPA-OS that provides services such as message passing, tasks and conversations.

AMS
Agent Management System is a mandatory part of the FIPA specified agent platform. AMS is responsible for maintaining agent lifecycle within the platform and the management of agents, such as creation and deletion of agents.

Control agent
Controls the link properties according to the FIPA Nomadic Application Support specification.

CORBA
Common Object Request Broker Architecture specified by OMG.

DF
Directory Facilitator is a mandatory part of the FIPA specified agent platform. DF offers yellow pages services for agents.

FIPA
Foundation for Intelligent Physical Agents (FIPA) is an international organization that aims to standardize an interoperable software agent architecture.

FIPA-OS
FIPA-OS is an open source software agent platform that complies with the FIPA specifications.

HTTP
Hypertext Transfer Protocol

IIOP
Internet Inter-ORB Protocol

IMTS
Intra-platform Message Transport Service. IMTS is responsible for internal communication within an agent platform. FIPA does not specify the internal operation of an agent platform.

iPAQ
Compaq iPAQ is a high-end PDA device.
| **J2ME** | Java 2 Micro Edition is an effort to provide a modular and scalable architecture for developing portable applications for resource constrained consumer and embedded devices. |
| **J2SE** | Java 2 Standard Edition is the traditional Java application environment. |
| **Kaffe** | Kaffe is a clean room implementation of the PersonalJava specification. |
| **Linux** | Linux is a Unix based open source operating system. |
| **MicroFIPA-OS** | The CRUMPET agent execution environment for small devices that is based on the FIPA-OS agent platform. |
| **Monitor agent** | Monitors the link properties according to the FIPA Nomadic Application Support specification. |
| **OMG** | Object Management Group is an international organization that specifies industry standards for object management. |
| **ORB** | Object Request Broker is a mandatory part of the CORBA architecture. ORB routes requests to remote objects. |
| **PersonalJava** | PersonalJava is a scaled down version of the standard Java 1.1 VM and contains only a subset of the JDK 1.1 APIs. The subset includes AWT with minor changes and classes that provide support for applets and applications. |
| **QoS** | Quality of Service |
| **RMI** | Remote Method Invocation is the Java way of executing distributed remote procedure calls. |
| **SL** | Semantic Language is a content language specified by FIPA. |
| **WAP** | Wireless Application Protocol |
1. INTRODUCTION

2. DESIGN RATIONALE

3. BACKGROUND

3.1 PERSONAL JAVA AND JAVA PLATFORMS

3.1.1 Overview of the Java Platforms

3.1.2 Java Limitations for Embedded Devices

3.1.3 PersonalJava

3.1.4 J2ME

3.1.5 Proprietary Solutions

3.2 LINUX FOR PDAS

3.3 THE FIPA ARCHITECTURE

3.3.1 FIPA Reference Model

3.4 FIPA-OS

3.4.1 High-level Architecture

3.4.2 Use of Profiles

3.4.3 CC/PP

3.5 QUALITY OF SERVICE (QoS) USING CONGESTION MANAGEMENT

3.6 SUMMARY

4. MICROFIPA-OS

4.1 DESIGN OVERVIEW AND RESTRICTIONS

4.2 MICROFIPA-OS COMPONENTS

4.2.1 Overview

4.2.2 Functionality of the Agent Shell

4.2.3 TaskManager

4.2.4 ConversationManager

4.3 FULL FUNCTIONAL AGENT PLATFORM

4.4 PARTITIONED PLATFORM

4.4.1 Message Transport Service (MTS) in a distributed FIPA-OS

4.4.2 Micro Agent Naming in a Distributed Agent Platform

4.4.3 Agent Startup

4.5 SOFTWARE PARTITIONING AND PROFILES

4.5.1 Partially Dynamic Partitioning of Agent Roles

4.5.2 Partitioning Using Profiles

4.5.3 Description for Partially Dynamic Partitioning

4.5.4 Partitioning ACC, AMS, DF and Service Agents

4.5.5 Making the Partitioning Decision

4.5.6 Downloading Code from the Network

4.5.7 CC/PP Profiles

4.5.8 FIPA-OS Profile Support

4.6 CONNECTION FAILURES

4.7 SUMMARY

5. CONCLUSIONS

APPENDIX A: JAVA PLATFORMS OVERVIEW

APPENDIX B: DYNAMIC PROFILES FOR FIPA-OS
2.1 Class Diagram .................................................................................................................. 22
2.2 Profile Implementations (e.g. ConversationManagerProfile) ........................................ 22
3 Generic Databinder Architecture ....................................................................................... 22

APPENDIX C: TASK MOBILITY .......................................................................................... 22

  Distribution of Tasks .......................................................................................................... 22
  UML Diagram: Basic Agent Relationships ....................................................................... 22
  Related Work .................................................................................................................... 22

APPENDIX D: LIST OF FIPA-OS OPTIMIZATIONS ............................................................. 22

APPENDIX E: IPAQ HARDWARE SPECIFICATION ............................................................. 22
1. Introduction

The overall aim of CRUMPET [CRU00] is to implement, validate, and trial tourism-related value-added services for nomadic users across mobile and fixed networks. In particular the use of agent technology will be evaluated (in terms of user-acceptability, performance and best-practice) as a suitable approach for fast creation of robust, scalable, seamlessly accessible nomadic services. The implementation will be based on a standards-compliant open source agent framework, extended to support nomadic applications, devices, and networks.

Nomadic users will access the CRUMPET services using so-called small devices like Personal Digital Assistants and Smart Phones. In order to implement the agent based services for these resource-constrained devices many design issues have to be taken into account. The amount of processing power and memory in small devices is restricted; quality of service and data throughput over different kind of links varies to great extent. All these issues and many others have to be coped with when the design of the CRUMPET system is done.

This deliverable concentrates on the design of CRUMPET agent platform for small devices. This forms the base for CRUMPET services and is the platform that the end-users of the CRUMPET system will use in order to interact with various tourism services on the network. This paper is organised as follows: section 2 presents the design rationale of the CRUMPET small footprint agent platform, and the agent shell. In this section we outline the major points in the design and give a brief discussion on the architectural issues examined in the later sections. Section 3 presents background information on topics that affect the design of the platform. This section discusses the Java platform, presents the open source FIPA [FIP00] compatible FIPA-OS [NOR00] agent platform, examines the Linux operating system on small PDA devices and presents a conclusion on these technologies and their role in the CRUMPET small footprint agent services. Section 4 presents an overview of the CRUMPET small footprint platform, the MicroFIPA-OS, and examines different design alternatives. The design goal is to keep the platform adaptable to different constrained environments, scalable and support the development and deployment of the CRUMPET services. Section 5 presents the conclusions of the design and summarizes the technologies that the small footprint CRUMPET platform, or MicroFIPA-OS, will be based on, such as the target device, operating system, Java platform, and the high level architecture that will be implemented. The development strategy of MicroFIPA-OS is seen as a continuous process of analysis and implementation, where the device performance issues and demands affect the final configuration of the CRUMPET system.
2. Design Rationale

The CRUMPET project uses software agent technology in order to create and deploy various tourism-based services for a mobile audience. The design goal of this document is to have a distributed agent platform that scales to a large number of nomadic users and supports the use of CRUMPET services on small devices. As the mobile devices are limited in processing power, memory and persistent storage we need to place agent platform functionality on systems on the fixed network to some extent. Issues like service discovery, profile distribution, content adaptation, and distribution of platform internals are issues discussed in this document.

Agent platforms in constrained environments have limitations in the number of threads, concurrency and memory. Therefore, very complex agents are not suitable for these environments; high number of parallel tasks, overhead in maintaining a large number of tasks, large amount of agent-to-agent high-level communication make them unsuitable. Agents for small resource constrained devices need to be lightweight and compact. We want to minimize the number of components running in parallel and reduce the high-level communication between these components.

The proposed designs in section 4 try to solve the problems related with middleware on resource-constrained devices by:

- The creation of pluggable system components for FIPA-OS that are optimized for constrained devices, especially for the Compaq iPAQ, but also other devices that support the PersonalJava programming environment. The work is done by extending the current Java 1.2 based FIPA-OS and using the existing automatic conversion to scale the work to Java 1.1. This gives us the leverage to support the future devices that have Java 2 support without considerable modifications to the platform implementation or design. We can support both platforms by using dynamic profile-based configuration information. For some components we may introduce designs that, in some ways, are limiting but perform better.

- Taking an advantage of fixed network agent platform services using the Access Node and proxies that are located on the fixed network. In addition, we also consider adding a "wrapper" layer to FIPA-OS to support distributed agent platforms. The key is to reduce the functionality needed to implement on the mobile host.

- Redesign FIPA-OS for small devices. This is the case if FIPA-OS does not scale to the constrained device environment and the system internals need to be re-designed and simplified. Issues that affect the re-implementation of any agent platform for constrained environments are: simplifying the platform API, reducing high-level communication between components and reducing low-level (method calls) communication between components and especially avoid unnecessary long method call trees.

The key idea behind the MicroFIPA-OS examined and presented in this document is to bring the benefits of agent technology to mobile users using various PDA-devices. The CRUMPET architecture is divided into three different parts:

1. Mobile terminals that run MicroFIPA-OS with PersonalJava. Mobile terminals are restricted in memory and processing power.

2. FIPA-OS Access Node (figure 1.I), which connects the mobile terminals into a single FIPA-OS agent platform domain. Alternatively in scenario II in figure 1 we may only have a federation of FIPA compatible agent platforms that the mobile devices, being also FIPA compatible platforms, may reference.
3. Service domains and platforms that provide the CRUMPET tourism-based services for the mobile terminals. Service platforms are FIPA compatible agent platforms.

In addition, we have two alternative scenarios to consider: a fairly large Access Node FIPA-OS domain that contains mobile nodes or each mobile device hosting a full FIPA platform (figure 1).

**Scenario I: Distributed Platform**

**Scenario II: Many FIPA Platforms Connected via ACCs**

Figure 1 Two possible scenarios for the CRUMPET platform architecture. In the first scenario FIPA-OS Access Node connects mobile devices running MicroFIPA-OS into a single FIPA-OS domain. Mobile clients access CRUMPET services through the Access Node. The second scenario features full FIPA compliant platforms on mobile devices. DF, AMS and ACC are mandatory FIPA platform components [FIP00]; IMTS is the internal message transport, which is outside the scope of FIPA.
3. Background

This section presents background information on Java and FIPA-OS. In section 3.1 we consider using the Java programming language and PersonalJava application environment on resource constrained devices. Section 3.2 presents the Linux operating system for PDAs and section 3.3 presents the FIPA architecture and 3.4 examines the open-source FIPA-OS agent platform. Section 3.5 outlines Quality of Service (QoS) using congestion management and section 3.6 presents the conclusions for the Java platform and operating system for the CRUMPET project.

3.1 PersonalJava and Java Platforms

3.1.1 Overview of the Java Platforms

The Java programming language has become the de facto standard for Internet programming. The key feature of Java has always been portability between different operating systems and software platforms. Originally, Java was developed to provide software for consumer electronics. Java also suited well into the distributed model of the Internet and has become the standard for Internet programming.

However, currently Java is not well suited for small embedded environments because of issues with speed, memory consumption, direct hardware control and programming, and scalability. Embedded environments have different requirements from the desktop world: different file-systems, ROM-based execution, and sometimes the requirement for real-time behaviour.

Currently there are several ways to use Java in creating software applications for embedded systems. Solutions come in two different categories: new specifications of the Java Application Environment (AE) and Virtual Machines for embedded computing and new implementations of existing specifications with embedded computing in mind. Figure 2 presents the different AEs that have been created for embedded application development. Of these the JavaCard AE is the smallest, aimed at the smart card sector. EmbeddedJava is a proprietary solution where the device manufacturers can choose what parts of the Java VM and class libraries to include in the application environment [Sun00d].

PersonalJava [Sun00a] is a scaled down version of the standard Java 1.1 VM and contains only a subset of the JDK 1.1 APIs. The subset includes AWT with minor changes and classes that provide support for applets and applications. The PersonalJava VM is an implementation of the Enterprise Java VM to resource-constrained devices.

In addition to the basic AEs described above, the Java 2 Micro Edition (J2ME) is an effort to provide a modular and scalable architecture for developing portable applications for consumer and embedded devices. For devices with strict system requirements J2ME provides minimal configurations of the Java virtual machine and Java APIs. For devices with new features these minimal configurations can be extended with additional APIs or more suitable virtual machines [Sun00e].
New Java specifications and solutions for embedded application development are:

- **PersonalJava**, supports consumer devices with a Graphical User Interface (GUI) and network connection. RTOS, set-top-boxes, PDAs.

- **EmbeddedJava**, supports devices with intermittent or no network connection and usually no GUI. EmbeddedJava can be configured to include only classes and VM features that are needed by a particular device. Controllers, printers, instrumentation.

- **JavaCard**, including the *iButton* implementation by Dallas Semiconductor, smartcards.

- **Java 2 Platform Micro Edition** (*J2ME*) with the *K Virtual Machine* (*KVM*) [Sun00b] are the latest additions. Both are used in the Motorola PageWriter 2000x, Palm III and Palm V hand-held devices. They provide *Jini* [Sun00f] support for small devices and allow dividing computing functionality between device and server.

In addition to AEs the Java functionality can be extended with Extension APIs, such as JavaPhone and JavaTV. Furthermore, a new runtime environment can be implemented and specified to improve support for thin clients, such as the *K Virtual Machine* (*KVM*). Moreover, many developers have their own implementation of the Java virtual machine, such as in the EPOC mobile operating system by Symbian [Sym99].

### 3.1.2 JAVA LIMITATIONS FOR EMBEDDED DEVICES

This section outlines the reasons why the standard Java specification and virtual machine are unsuitable for embedded application development.

**Hardware Programming**

It is impossible to write device drivers in Java, because it does not support direct memory access (pointers) and interrupt handling. Java programs cannot directly reference hardware devices and ports.
However, it is essential for embedded software to utilise the special hardware of the device. With Java this can be accomplished using the standardised Java Native Interface (JNI) [WRi99]. Native methods must be implemented with another programming language, for example C or C++.

**ROM-based Execution of Java Classes**

Usually workstations execute Java bytecode by either downloading it from a computer network or loading it from the local storage. Class files are loaded into RAM; the bytecode is verified and then executed by the JVM. Now, in many cases embedded devices need to execute the bytecode from ROM. A ROMizer utility is needed to execute Java code from ROM. For instance the CodeCompact, which processes the Java class file into an appropriate run-time format that the JVM can execute from ROM or flash memory [WRi99]. The CodeCompact is a utility tool that structures the Java bytecode class files for ROM and flash memory based systems. It is not necessary to perform class-file loading and bytecode-verification phases when using the ROMizing process. As such the process improves Java application start-up time.

In essence, classes are “pre-loaded” as part of the process of making the ROM image for a particular device. For instance, this is used in the EPOC system. The preloader generates the run-time structures. The preloader can also try to share as many of the generated structure as possible to optimise memory usage of the classes within ROM. However, preloaded classes cannot be overridden by another version of the class. This effectively makes it impossible to tamper with preloaded classes, but makes updating also impossible [Sym99].

**Real-time Requirements**

Many embedded devices have requirements for real-time data processing. Java’s non-deterministic garbage collection makes it unfit for real-time operation. The Java garbage collection must be finished once it starts and cannot be pre-empted by a high priority thread. At the time of the writing there are no real-time Java implementations compatible with Java platform specifications [WRi99]. One solution is to develop the real-time part in C or C++ and use JNI to connect the real-time component to non-real-time Java-component.

### 3.1.3 PERSONALJAVA

The PersonalJava technology consists of two parts: the PersonalJava Application Programming Interface (API), and the PersonalJava virtual machine. When implemented these parts form the PersonalJava Application Environment. PersonalJava is optimized for consumer electronic devices, such as Internet telephones, personal electric organizers, and set-top boxes. The basic idea behind the optimizations in PersonalJava is that most consumer devices do not need the functionality of the full JDK specification (figure 3).

Sun has released the JavaCheck tool that can be used to verify an application’s conformance to the PersonalJava API specification. Moreover, PersonalJava applications can be developed on desktop computers without executing the software on the target device. The PersonalJava emulation environment allows applications to be executed and tested. A beta version of the PersonalJava runtime environment is available for download for Windows CE operating system [Sun00a]. In addition, the source code for the PersonalJava 1.2 specification is also available.
<table>
<thead>
<tr>
<th>Packages</th>
<th>Java</th>
<th>PersonalJava</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>Modified: .awt,.io,.text,.zip</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optional: .security,.math,.sql,.rmi</td>
</tr>
<tr>
<td>Graphics</td>
<td>Full AWT (X11,Motif, Windows)</td>
<td>Reduced AWT Optional windowing Low-level 2D native graphics stack.</td>
</tr>
</tbody>
</table>

Figure 3 Differences between Java and PersonalJava [Sun00a].

3.1.3.1 Optimizations and Features

The PersonalJava technology introduces a number of optimizations to the standard Java implementation. The standard Java classes have been re-implemented. The public interfaces ensure interoperability, although the actual code implementation has changed due to optimizations. Most of the standard Java functionality is retained, such as threads. One goal has been to reduce static memory footprint and runtime memory usage of the virtual machine, which is more compact. The Java CodeCompact tool is used to preload core classes into an alternate format that requires mostly ROM and a small amount of RAM.

In PersonalJava some packages are optional, which reduces the minimum amount of space required for the application environment. Furthermore, lossless compression with Java Archive (JAR) can be used to further reduce memory requirements and make data transfer from the network more efficient. Optional packages include packages pertaining to RMI (Remote Method Invocation), security and SQL. In addition, PersonalJava contains an optimized version of the AWT (Abstract Windowing Toolkit) for displays limited in size and colours.

3.1.3.2 Consumer Graphical User Interface

The user interfaces of consumer devices differ from the interfaces used in mainstream desktop systems. Nomadic devices utilize a wider range of input options. In addition, output options are also versatile, but more limited than their desktop counterparts. Finally, user interfaces for consumer/nomadic devices need to be simple and easy to use.

PersonalJava contains a subset of the JDK’s Abstract Windowing Toolkit (AWT) that is specifically tailored for consumer device requirements. The use of complex interaction components, such as layered menus and horizontal scrolling, is de-emphasised. The Personal AWT provides support for a variety of input devices, such as a virtual keyboard, a joystick, specialized buttons, a pen, a remote control, a touch screen or speech.

Several factors affect GUIs in embedded devices:

- The screen size is smaller than in desktop Java applications.
- The screen may not support colours.
- The input device is probably proprietary.
- RAM, ROM, battery and performance are limited.
- AWT-implementation is not identical to the desktop counterpart. Buttons, boxes and most importantly scrollbars are placed differently and have different characteristics.

### 3.1.3.3 PersonalJava and J2ME

The current generation of the PersonalJava application environment [Sun00a] is based on a JDK 1.1 code base. The next release of PersonalJava technology will be based on a Java 2 Platform code base and be separable into components: the Connected Device Configuration (CDC) and the Personal profile which will provide full compatibility for applications written to the PersonalJava runtime specification.

### 3.1.4 J2ME

This section presents the Java 2 Micro Edition, which is a set of Java language and virtual machine specifications for different constrained environments. The J2ME consists of configurations, which provide the basic foundations and profiles that provide additional features and complement the API.

#### 3.1.4.1 Overview

J2ME is a modular architecture (figure 4) that provides minimal configurations of the Java virtual machine and Java APIs. J2ME contains a small core API, which must be implemented in every J2ME compatible device. Each configuration addresses a particular class of device and specifies a set of APIs and Java virtual machine features that are present on the devices. Mandatory components are from java.io, java.net, and java.lang packages.

Application development with J2ME is done on workstations using standard Java development tools. After compilation, the Java class files are converted to support the needs of a particular application profile. The converter optimises the bytecode to fit within the small footprint of an embedded device. This conversion process can include converting Java class loading to support ROM-based loading.

Java 2 Micro Edition currently includes two different implementations (configurations) of the Java virtual machine. J2ME consists of two configurations [Sun00c, Sun00e]:

1. **CDC (Connected Device Configuration)**, 32-bit VM and 512KB of ROM. A full Java virtual machine for 32-bit RISC/CISC/DSP microprocessors and a few megabytes of working memory. Typical products are screen phones, set-top-boxes and high end PDAs.

2. **CLDC (Connected Limited Device Configuration)**, which uses 16- or 32-bit KVM and 256-512KB of memory. Does not necessarily require a persistent network connection. Uses the K virtual machine that targets 16-bit or 32-bit RISC/CISC microprocessors with a few hundred kilobytes of memory. Typical products are cell phones, low-to mid-range PDAs and low-end set-top boxes.

Figure 5 presents the J2ME hierarchy. From the bottom we have the native OS, which hosts a suitable Java virtual machine. The VM may be a full VM adhering to the standard specification [Kra99] or support only some subset of the full specification, for example the KVM [Sun00b]. Above the VM we have the class libraries (packages) that provide the basic API for applications. Again, they may be able to support the Java 1.2 packages or only some subset of them. The configurations define the VM and the supported core Java libraries (such java.lang and java.util). Profiles are above the configurations and define aspects pertaining to vertical markets. Configurations define generic interfaces (such as the generic connection framework in CLDC described in section 3.1.4.3) and profiles define the actual protocols that are available and, for example, user interface issues.
Sun and industry partners define and develop J2ME and its configurations through the JCP (Java Community Process). Profiles are implemented within these configurations and each targets a vertical market and provides domain-specific classes (details the APIs). CDC vertical profiles target television and automotive applications. However, CDC is currently under work and has not yet been officially released. The foundation profile in figure 4 is defined to provide the full Java 2 API for the CDC configuration. PersonalJava will be integrated into the J2ME as a personal profile (figure 4). EmbeddedJava is also incorporated into J2ME. EmbeddedJava provides the Java technology benefits, but does not require a platform-based solution. In addition, a RMI profile is under work as a JCP, but a beta has not yet been released.

Current CLDC profiles target handheld devices and wireless phones. The only profile currently available for CLDC is the Mobile Information Device Profile (MIDP) that includes a reference implementation for Windows and Solaris [MID00]. CLDC source-code is available under the Java Community Process source license. A runtime of J2ME CLDC is available for PalmPilot devices (III, V, VII), which can be used to develop Java applications [Sun00b].
Java 1.1 Platform
Sun Standard Edition 1.1 VM
- AWT, RMI, JIT, JNI, CORBA, RMI-IIOP

Kaffe (PocketLinux and many other systems) is a clean room implementation of 1.1 JVM (PersonalJava) (no license required).
- AWT, Swing support
- Minimum build 192kb
- JNI
- Beta support for RMI and no support for Java security

EPOC Java
- JDK 1.1.4
- 128 k
- JNI, RMI, AWT, Swing

Java 1.2 Platform
Sun Standard Edition 1.2 VM
- AWT, SWING, RMI, JIT, JNI, CORBA, RMI-IIOP

Java 2 MicroEdition (J2ME)

Configurations

CDC (Connected Device Configuration)
- PDAs
- Standard edition JVM

CLDC (Connected Limited Device Configuration)
- smart phones
- KVM (80k footprint)

Profiles

Personal
- Support for PersonalJava

MIDP (Mobile Information Device Profile)
- sockets as baseline
- not necessarily IP-stack (also WAP...).
- 128-512 KB (RAM+ROM)

RMI profile
- currently developed as a Java Community Process
- requires the foundation profile

PDA profile
- for more efficient devices than MIDP
- currently developed as a Java Community Process
- 512 KB – 16 MB for Java runtime

Foundation profile
- entire Java 2 API

Figure 4 Overview of the J2ME.
3.1.4.3 CLDC

The CLDC specification addresses the Java language and virtual machine features and defines the core Java libraries (java.lang.*, java.util.*), input/output features, basic networking with the Generic Connection Framework, security and internationalisation. CLDC configuration does not include application life-cycle management, user interface functionality, event handling or a high-level application model. These features are left to be defined in profiles, which complement a J2ME configurations’ API.

The goal of CLDC’s Generic Connection Framework is to be a functional subset of the J2SE implementation, and provide better extensibility and flexibility in supporting new devices and protocols. The system supports input/output and networking in a general framework that map to common low-level hardware. All connections are created using a single static method in a system class called Connector. This method returns an object that implements one of the generic connection interfaces.

CLDC itself does not include any protocol implementations. Connectors may support, for example, http, sockets, communication ports, datagrams and files. The key benefit of this design is that the implementation of the application is the same, even though the connection type is changed. The binding of protocols to a J2ME program is done at runtime. This means that the connection type (string given by the programmer as an argument to the Connector method) instructs the system to load the proper protocol implementation. This is called late binding in the CLDC specification, and this procedure permits a program to dynamically adapt to use different protocols at runtime. In addition, a hierarchy of Connection interfaces that group together classes of protocols with the same semantics.

Depending on the capabilities of the device the CLDC may support multiple simultaneously running Java applications. CLDC supports also dynamically downloading Java classfiles. Software is publicly distributed as JAR files that contain preverified classfiles.

![Figure 5 J2ME High level architecture.](image)
Summary of CLDC limitations:

1. Sandbox security model.
2. No floating point arithmetic and support for multidimensional arrays.
3. No implementation of Java Native Interface (JNI). Wrappers to native code have to be done using implementation specific means.
4. No custom class loaders.
5. No support for reflection or thread groups or daemon groups.
6. Point 5 means that a JVM supporting CLDC does not support RMI, object serialization, JVMDI (Debugging Interface), JVMPI (Profiler Interface) or any other advanced features of J2SE that depend on the presence of reflective capabilities.
7. Class files that support CLDC need to be preverified.
8. Regular Java classfiles are not optimized for network transport in bandwidth-limited environments. This is because each Java classfile is an independent unit that contains its own constant pool (symbol table), method, field and exception tables, bytecodes, and some other information. A future file format could improve this if full symbolic information was left out.

MOBILE INFORMATION DEVICE (MID) SPECIFICATION

Mobile Information Devices (MID) span a potentially wide set of capabilities. Rather than try to address all such capabilities the Mobile Information Device specification (MIDP) [MID00] addresses only those APIs that were considered absolute requirements to achieve broad portability. These APIs are: application (the semantics of a MIDP application), user interface (includes display and input), persistent storage and networking. From the networking point of view, HTTP 1.1 is the baseline requirement for MIDP, because the mobile information device may have no support for the Internet Protocol. A MIDP application is called as a MIDlet and the MIDP high-level architecture is portrayed in figure 6. A MID should have the following minimum characteristics:

- Screen-size: 96x54
- Display depth: 1-bit
- Pixel shape (aspect ratio): approximately 1:1
- One or more of the following user-input mechanisms: one-handed keyboard, two-handed keyboard, or touch screen.
- 128 kilobytes of non-volatile memory for the MIDP components.
- 8 kilobytes of non-volatile memory for application-created persistent data.
- 32 kilobytes of volatile memory for the Java runtime (e.g., the Java heap).
- Support for two-way, wireless and possibly intermittent connection with limited bandwidth.
Figure 6 High-level architecture. Mobile Information Device Profile (MIDP)

KVM

The K Virtual Machine (KVM) [Sun00b] is a new implementation of the Java virtual machine specification targeted at devices with limited processing capacity and memory capacity around 128 K (from 80 to 100 KB). KVM is part of the Java 2 Platform Micro Edition (Java 2 ME) architecture. The following points were the main focus of the KVM:

- Optimizations for small size (footprint)
- Easily portable to different platforms
- Modular and extensible
- Source written from scratch in C with special optimizations
- With minor exceptions identical to the Java virtual machine specification.
- Native support for I/O and graphics (no support for AWT or Swing). Uses wrappers and native functions to call host system functions instead of Java native interfaces (JNI).
- To produce a KVM capable of supporting consumer Jini technology client mode.

The KVM technology is available through Sun's Community Source License.

3.1.5 PROPRIETARY SOLUTIONS

3.1.5.1 Kaffe

Kaffe is a clean room implementation of the PersonalJava specification [Sun00a] (JDK 1.1.6). Kaffe is optimised for small devices and requires a footprint of 54 KB minimum for the JVM and 138 kilobytes for the core libraries. Kaffe supports AWT, Swing, JNI, but does not implement the optional packages that provide support for RMI and security. Kaffe is currently available for download for Linux and many other operating systems and the source code is also available (www.transvirtual.com).

Kaffe has been designed with customization in mind and the interfaces to the operating system, class loaders, file system, network and graphics sub-system are quite portable. The portability of Kaffe has led to ports of Kaffe for various operating systems and hardware platforms. Kaffe also supports native operating system threads and has an internal thread management system.
The Kaffe distribution from Transvirtual comprises of a full implementation of the PersonalJava class libraries: java.lang, java.io, java.util, java.net, java.beans, java.awt, java.applet, java.lang.reflect, java.util.zip, java.awt.datatransfer, java.awt.event and java.awt.image. AWT support is also provided for non-windowing systems that have no framebuffer with a custom Java based windowing system. Kaffe also works in a diskless environment and does not need a native file system. Java classes can be downloaded from RAM, ROM or flash memory.

3.1.5.2 Jeode

The Jeode Java platform from Insignia (www.insignia.com) is a PersonalJava and EmbeddedJava implementation based on JDK 1.1.7b. Jeode supports JavaBeans, RMI and is available for Linux, Windows, Windows CE and a number of other platforms.

3.1.5.3 EPOC Java Implementation

The EPOC Java implementation [All00] is based on JDK 1.1.4 and supports one VM per EPOC process. The EPOC web browser can execute multiple applets in the same VM, while each Java application runs in a separate EPOC process. In the implementation one EPOC thread is assigned per Java thread. The EPOC VM supports the basic JDK 1.1.4 features: AWT, RMI, JavaBeans and JDBC APIs. The EPOC Java development environment comprises of the EPOC emulator for Windows NT4/95/98, documentation in HTML format, tools to support native method development using JNI and a set of tools for deploying application on EPOC devices.

Running Java on EPOC devices requires at least a 32-bit CPU, 36 MHz clock speed or better, although some simple applications may operate at speeds as low as 18 MHz [All00]. The memory requirements are below 2 MB for small Java applications with a GUI. ROM footprint of the VM and class libraries is 2.7 MB. EPOC VM Caffeine benchmark results are presented in table 1. The EPOC Java VM seems to perform better than some other Java VM implementations. The VM in Psion netBook, according to table 1, in considerably faster than the VM in Jornada 820 which runs Windows CE [All00].

<table>
<thead>
<tr>
<th>Device</th>
<th>Processor</th>
<th>CaffeineMark 3 Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Psion Series 5MX</td>
<td>36 MHz ARM710T</td>
<td>26</td>
</tr>
<tr>
<td>Psion netBook</td>
<td>190MHz StrongARM SA11100</td>
<td>141</td>
</tr>
<tr>
<td>Desktop PC without JIT, 128MB</td>
<td>366 MHz Pentium II</td>
<td>500</td>
</tr>
<tr>
<td>Desktop PC with JIT 128 MB</td>
<td>366 MHz Pentium II</td>
<td>5000</td>
</tr>
<tr>
<td>Jornada 820</td>
<td>190MHz StrongARM SA11100</td>
<td>27-28</td>
</tr>
</tbody>
</table>

Table 1 Benchmark results for a number of devices using CaffeineMark 3 [All00].

In the next release (EPOC 6) the Java VM will support the PersonalJava (JDK 1.1.6), which is essentially same as the JDK 1.1.4, but supports RMI over IIOP. PersonalJava requires less memory than JDK 1.1.4 and hosts a set of improved tools for application development and deployment.
3.2 Linux for PDAs

The Linux operating system was created by Linus Torvalds at the University of Helsinki in 1991 and since then the free operating system has become one of the fastest growing platforms. Linux on PDA devices offers several advantages to other, possibly commercial, operating systems that ship with devices. Most importantly, Linux source code is available so developers can trim the operating system and tailor it for the constrained environments found in small devices. Linux can be tailored for different purposes. In addition, since embedded software development is usually done using cross-compilation, Linux and GCC for example offer a development environment that runs across a wide variety of hardware platforms.

Linux provides the basic networking functionality such as IP stack, availability of device driver source and also an existing software base that comprises of a huge variety of different applications and libraries. The Linux kernel provides support for memory management, process and thread creation, interprocess communication, interrupt handling, execute-in-place ROM filesystems, RAM drives and flash management. Figure 7 presents an overview of the different software and communication layers needed in the small device for the CRUMPET project. From bottom up we have the Linux kernel, which provides the basic operating system features, networking and device drivers. Then we have the X-Windows graphical windowing environment and either at the same level or on top we have the Java virtual machine (Kaffe), which may provide access to hardware through the JNI interface. The Java virtual machine executes the agent platform, which in turn hosts the agents.

On the downside, Linux support in PDAs is currently under development and is not commercially supported at the present moment. However, the first Linux based PDAs are beginning to become available. For example, Compaq maintains the www.handhelds.org website, which promotes and supports the use of Linux on Compaq PDAs. Moreover, Linux is currently being used in routers and set-top-box type devices.

The current embedded Linux packages use various tools to automate the Linux configuration, employ different GUIs and windowing environments which vary in size, features and appearance, use different drivers for a wide range of embedded applications. However, all the solutions share the common core that consists of the kernel.

The X Windowing System that usually provides the graphical windowing environment in Linux is a client/server architecture where UNIX applications running on any host can display an X-based terminal. Today both the applications and the X-servers run on the client machine. However, the client/server paradigm is not necessarily well suited for the embedded devices if the server needs to be active on the device. If the graphics needs of the applications are modest and the device is very limited we need to have other solutions. MicroWindows is one attempt to create a windowing environment that complies with the Microsoft WIN32 GDI API and XLIB.

PocketLinux differs from most embedded Linux projects, because it focuses on Java based applications and the heavy use of XML. There are several other projects using Linux in small devices, both commercial and open-source, but they do not rely so much on Java. Kaffe has been ported to many devices and embedded operating systems, and the Kaffe that comes with PocketLinux is another variant tailored for the StrongARM and MIPS processors.

The PocketLinux distribution is available for iPAQ, Helio, Cassiopeia (E15,E105,E115) and Philips Velo from Transvirtual (www.transvirtual.com). PocketLinux consists of Linux 2.4.x kernel re-engineered for PDAs, Kaffe, and support for software profiles in XML. The software package is distributed under the GNU GPL software license. Installation of PocketLinux to iPAQ or Cassiopeia may be a bit difficult, because the Linux kernel has to be first transferred to the device via a serial cable. After the boot loader and kernel is in place, software such as Kaffe can be installed. Currently,
it may be possible to retain the PocketPC operating system, which ships with the devices. Linux supports PCMCIA but PocketLinux support is forthcoming (already demonstrated). Furthermore, PocketLinux supports MP3 and several movie formats. One of the advantages of PocketLinux and iPAQ is the PCMCIA extensibility. This combination supports the use of wireless LAN cards, for example.

The PocketLinux binaries include a trimmed down version of Kaffe 1.0.6 (adheres to PJava 1.1.1 specification, JDK 1.1.7). The package contains a very simple emulation environment that available for Debian and Red Hat (only applets and network support requires recompilation). Red Hat is the suggested development environment when creating software for the PocketLinux and Kaffe. The PocketLinux Kaffe class libraries may be used for cross-platform development. In addition to the simple emulation environment, there is also a real Helio emulator for Win32. However, Helio has less performance than the other devices and is available in greyscale only.

Kaffe can be tailored to some extent to suit different needs. The default binary distribution of Kaffe for PocketLinux contained the following packages: lang, lang.reflect, awt, io, net, util, util.zip, text, applet. Source build included additional classes and packages: VideoPane in kaffe.awt, kaffe.media (MP3 codec), kaffe.util. IntegerHashtable, kaffe.util. Queue, kaffe.util. Timer. The video and sound capabilities cannot be tested on the Applet based emulator.

PocketLinux supports also Flash, the .swf file format, and MPEG video compression. The configuration of the basic application suite, which consists of calculator, calendar, notepad, clock, agenda, shell, email application, videoplayer and mediaplayer, is configured using XML files. PocketLinux Kaffe supports XML parsing and the W3C DOM model. In addition, Jabber, which is an instant messaging program, comes with the distribution. The PocketLinux Java applications use the W3C DOM package org.w3c.dom, which is part of the distribution.

![Figure 7 Overview of the different software layers needed for the CRUMPET software architecture on PDAs.](resource-url)
3.3 The FIPA Architecture

Foundation for Intelligent Physical Agents (FIPA) is an international organization that aims to standardize interoperable software agent architecture. The FIPA architecture can be divided into three phases: the FIPA97, FIPA98 and currently the so-called FIPA2000 specifications [FIP97,FIP98,FIP00]. The FIPA97 specification introduced the FIPA compatible agent platform, the agent life cycle, agent communication and agent management. FIPA does not specify the internal structure of agents or agent platforms, but concentrates on the outer interfaces and emphasises interoperability of agent systems. FIPA 98 introduced improvements to the 97-specification suite, such as human-agent interaction and security. However, since 97 and 98 the nature of the specification process has changed with the introduction of FIPA document policy. Currently, FIPA specifications are classified into preliminary, experimental, standard, obsolete and deprecated documents. Currently, most FIPA97 and FIPA98 specifications have the obsolete status, and most of the new FIPA2000 specifications are either preliminary or experimental.

3.3.1 FIPA Reference Model

The FIPA reference model shown below in figure 8 illustrates the core components of the FIPA architecture and also the FIPA-OS agent platform discussed in section 3.4. The agent reference model provides the normative framework within which FIPA Agents exist and operate. Combined with the Agent Life cycle, it establishes the logical and temporal contexts for the creation, operation and retirement of Agents [FIP00c].

![FIPA-OS Agent Framework](image)

Figure 8 FIPA and FIPA-OS Agent Framework and the FIPA specified platform agents.

The Directory Facilitator (DF) and Agent Management System (AMS) are specific types of agents, which support agent management, and the Agent Communication Channel (ACC) is a lower-level entity that is part of the MTS (Message Transport Service). The DF provides "yellow pages" services to other agents. The AMS provides platform management functionality, such as monitoring Agent lifecycles and ensuring correct behaviour of entities within, and upon, the platform. The ACC supports interoperability both within and across different platforms; therefore, it is viewed as a component of the MTS. The MTS provides a message routing service for agents on a particular platform. Such Agents must be reliable, orderly and adhere to the requirements specified in the FIPA MTS Specification [FIP00b].

The AMS, MTS and DF form what will be termed the Agent Platform (AP). These are mandatory, normative components of the model.
FIPA requires that the platform agents are able to receive ACL messages (Agent Communication Language) and can reply at least with the not-understood communicative act. Agent communication in the FIPA standard is based on the speech act-theory and agents try to influence others using different performatives (communicative acts), such as request or inform. In order to simplify this high level communication FIPA uses various interaction protocols that specify the possible states of the conversation. The FIPA platform agents must support the SL0 content language and the Agent Management ontology, which includes the basic platform functionality. Content languages are used to represent the contents of an ACL message.

The requirement for the platform agents to be located in the small foot-print device necessitates the use of the SL0 content language, ACL-messaging and support for the fipa-agent-management ontology on the small device. In addition to SL0, we need to have different parsers for other content languages (such as XML), if we use any.

Message Transport Model

The FIPA Message Transport Model (MTM) consists of Message Transport Protocols (MTP), which carry out the physical transfer of messages between two ACCs. In addition, we have the Message Transport Service (MTS) provided by the ACC, which supports the transportation of FIPA ACL messages between agents on any AP. In essence, the transportation of messages is a service provided by the ACC. ACC is only required to read the message envelope, it does not need to parse the contents of the message. The internal representation of messages and interaction between ACC, AMS and DF are left to the developer, FIPA specifies only the outer interfaces to other platforms.

Agent Relationship with AP

The FIPA focuses on the agent platform abstraction and leaves the internals of the platform and agents to developers. An agent, developed using any programming language, needs to have the capability to formulate ACL messages and to contact the home platform internals, the DF, ACC and AMS. How the agents communicate with the internal platform components is an implementation issue and not specified by FIPA. However, all agents should be able to receive and send ACL messages, and be able to reply at least with the communicative act of not-understood.

Agents may send messages using the ACC to other platforms. The ACC forwards incoming messages to either local agents or foreign ACCs. In order to receive messages, the ACC of the platform needs to forward the messages to the proper transport address. This is internal functionality of the platform and FIPA does not specify this. In general, FIPA agents have three ways of communication: first they can send the message to the ACC for forwarding to a foreign ACC, then they can directly send the message to the foreign ACC if they know its address, and finally they can communicate directly using a proprietary non-FIPA means.

Intra-platform and Inter-platform Communication

When compared with the FIPA97 standard the newer FIPA specification drafts (of which none has yet reached the official standard status) offer a more abstract setting with the Abstract Architecture specification. With the newer specifications we do not necessarily have to use IIOP in inter-platform communication. We can use whatever protocol we like. However, the current FIPA Message Transport draft specification implies that IIOP, WAP or HTTP should be supported for interoperability. The required support for inter-operability (IIOP) with the older FIPA97 agent platforms can be provided in the form of gateways between platforms.

FIPA-OS supports already IIOP for inter-platform communication and this can be used also on the small device if an Object Request Broker (ORB, such as the HORB) [HOR00, OMG00] is present on
the system. Since Java 1.2 or even 1.1.8 is not available for many small devices additional ORB is needed. In addition, other possible transports for the small footprint agent platform are WAP and HTTP. HTTP is more readily available in small devices than IIOP, but does not offer reliable message transport. If new transports are required, the FIPA-OS transports can be extended with the desired transport protocol. The communication between the CRUMPET Access Node and the MicroFIPA-OS is proprietary, and FIPA specified ACL messaging is mainly needed when the platform needs to talk to platforms without using the Access Node.

The working hypothesis in the current design is that all communication is in FIPA ACL, and that the QoS of the wireless transport needs to be monitored and the link controlled. The monitoring of the QoS is done by the Monitor agent and the Control agent makes decisions about controlling the use of the wireless link. Control and Monitor agents are specified and implemented by WP3. There are several problems related with the controlling of the QoS. First of all we need priority classes for packets, and also need to have flow control for the streams. The lower priority traffic streams should not override higher priority streams and the sender should not flood the receiver with data.

3.4 FIPA-OS

FIPA-OS is an open source implementation of the mandatory elements contained within the FIPA specification for agent interoperability. In addition to supporting the FIPA interoperability concepts, FIPA-OS also provides a component-based architecture to enable the development of domain specific agents that can utilise the services of the FIPA Platform agents.

The primary aim of FIPA-OS is to reduce the current barriers in the adoption of FIPA technology by supplementing the technical specification documents with managed open source code. It has already been seen that the quality and functionality of this source code will improve by managing its evolution in the public domain, providing mutual benefit to all adopters and enabling progress in the agent paradigm. Developers using FIPA-OS are encouraged to provide extensions, bug fixes and feedback to help improve the planned future releases.

FIPA-OS has been implemented using Java 1.2, and the distribution contains class and source files and documentation. It also includes simple test agents to access the agent platform services and some visualization software. Currently, a JDK 1.1 version of FIPA-OS is also available.

FIPA-OS handles the initial bootstrapping required to enable agents on multiple, potentially heterogeneous, FIPA Agent Platforms to interoperate via the use of a web server. The ACC included in the current FIPA-OS distribution uses the web servers that it knows about to obtain the required MTP (Message Transport Protocol) addresses for the initial platforms with which it requires interoperability. The interaction with the web servers is performed at start-up only and not when each message is routed to a remote Agent Platform. In the situation where the ACC has to be restarted it uses an inform message to notify all known ACCs of its new MTP addresses. Likewise, the ACC assumes that remote ACCs will notify it of their new MTP addresses should they be changed due to them being re-initialised.

3.4.1 HIGH-LEVEL ARCHITECTURE

FIPA-OS is a component-orientated toolkit for constructing FIPA compliant Agents using mandatory components (i.e. components required by ALL FIPA-OS Agents to execute), components with replaceable implementations, and optional components (i.e. components that a FIPA-OS Agent can optionally use). Figure 9 below highlights the available components and there relationship with each other.
The Database Factory, Parser Factory and CCL components are optional and do not have an explicit relationship with the other components within the tool-kit. The Planner Scheduler generally has the ability to interact with all components of an Agent, although not necessarily vice versa.

The replaceable implementations included as part of the FIPA-OS distribution for each component include:

- MTPs
  - RMI (proprietary)
  - IIOP (FIPA compliant)
- Databases
  - MemoryDatabase
  - SerializationDatabase
- Parsers
  - SL
  - ACL
  - XML
  - RDF

The Agent Shell that the user extends when building agents provides the following functionality:

- Sending messages
- Retrieving the agents’ properties (Profiles, AID, state) & locating platform agents (DF and AMS)
- Registration with platform agents
- Setting up Tasks
- Shutting down the agent (the agent and its components can/should be cleanly shutdown)
3.4.2 Use of Profiles

FIPA-OS relies heavily on use of profiles. The profiles provide an easy way to configure agents’ runtime without having to recompile the code; agents can also change the profiles. At the moment, there are two types of profiles in use: the platform profile and the agent profiles, all the profiles are encoded in RDF/XML. The appendix B presents a detailed technical design of improving the current FIPA-OS profiles towards more multi-purpose profiles that can be used to store, for example, device configuration information.

The platform profile describes the common properties of all the agents on the platform – all the agents always read this profile when starting up. The profile contains information like location of the other profiles, name of the platform and the name of the AMS.

The second types of profiles are the agent profiles. These profiles contain information related to individual agents, and they are named after their owners (like AMS’ profile would be called ams.profile). Agent profile should contain information about the platforms internal transport address, and specify the database type and location that the agent wants to use. It has been noticed though, that standard agents don’t need any specialised information and therefore FIPA-OS provides a special type of agent profile called default profile (and named default.profile) that agents use if their individual profiles are not provided.

FIPA-OS profiles are extremely flexible and extensible. As can be seen from the last example, from the agent profile (AgentProfile) and reference is made to a ‘Database’ object (<ns1:Database rdf:resource="database1"/>), which is then presented next. However, there is no reason why this object couldn’t be in a separate physical file, that could potentially be located in the network.
The existing FIPA-OS profiles could be extended to contain additional agent configuration information, such as preferences associated with the delivery of content. Work on this kind of profiles has that can be used to expand FIPA-OS has been done by Composite Capabilities/Preference Profiles (CC/PP) work group. Arbitrary XML DTDs or RDF Schemas can be incorporated to further expand the configuration ability of the agent. The agent itself would determine how the configuration information is used for domain specific purposes - i.e. separating out what configuration information is handled by default from what configuration information would have to be handled in a special way within the context of CRUMPET. Agent could, for example, have a hardware profile written in CC/PP, like the example below.

```xml
<?xml version="1.0" encoding="UTF-16" ?>

<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:ccpp="http://www.w3.org/2000/07/04-ccpp#"
     xmlns:uaprof="http://www.wapforum.org/UAPROF/ccppschema-19991014#">
  <Description about="http://www.example.com/MyProfile">
    <ccpp:component>
      <Description about="http://www.example.com/TerminalSoftware">
        <type resource="http://www.example.com/Schema#SoftwarePlatform" />
        <uaprof:OSName>EPOC</uaprof:OSName>
        <uaprof:OSVersion>2.0</uaprof:OSVersion>
        <uaprof:OSVendor>Symbian</uaprof:OSVendor>
      </Description>
    </ccpp:component>
  </Description>
</RDF>
```

### 3.4.3 CC/PP

CC/PP is a W3C work group whose mission is to develop an RDF-based framework for the management of device profile information [W3C00a]. The group has produced working drafts as part of the W3C Mobile Access activity that incorporates suggestions resulting from reviews and active participation by members of the IETF CONNEG (Content Negotiation) [IET00] working group and the WAP Forum UAprof drafting committee [WAP00].

CC/PP is a collection of the capabilities and preferences associated with user and the agents used by the user to access the World Wide Web. These user agents include the hardware platform, system software and applications used by the user. User agent capabilities and references can be thought of as metadata or properties and descriptions of the user agent hardware and software.

A description of the user's capabilities and preferences is necessary but insufficient to provide a general content negotiation solution. A general framework for content negotiation requires a means for describing the metadata or attributes and preferences of the user and his/hers/its agents, the attributes of the content and the rules for adapting content to the capabilities and preferences of the user. The current mechanisms, such as accept headers and `<alt>` tags, are somewhat limited. Future services will be more demanding. For example: the content might be authored in multiple languages with different levels of confidence in the translation and the user might be able to understand multiple languages with different levels of proficiency. To complete the negotiation some rule is needed for selecting a version of the document based on weighing the user's proficiency in different languages against the quality of the documents various translations.
Some potentially complex negotiation may have to take place between the content or the server of the content and the user of the content. Though it is envisaged that the use of RDF to encode the metadata describing the content and the user's preferences will facilitate the development of solutions to these kinds of complex negotiations, the implementation of appropriate rules for the negotiation is left to application developers.

The goal of the CC/PP framework is to specify how client devices express their capabilities and preferences (the user agent profile) to the server that originates content (the origin server). The origin server uses the "user agent profile" to produce and deliver content appropriate to the client device. In addition to computer-based client devices, particular attention is being paid to other kinds of devices such as mobile phones.

The requirements on the framework emphasize three aspects: flexibility, extensibility, and distribution. The framework must be flexible, since we can't today predict all the different types of devices that will be used in the future, or the ways those devices will be used. It must be extensible for the same reasons: It should not be hard to add and test new descriptions. And it must be distributed, since relying on a central registry might make it inflexible.

The CC/PP work group has also defined CC/PP vocabularies. The vocabularies are used to describe the attributes of clients, and of proxies that offer content adaptation possibilities. The work group has discussed the types of values to which CC/PP vocabularies may refer, a description of how to define new vocabularies, a small client vocabulary covering print and display capabilities, a vocabulary for proxies and other intermediaries to describe additional capabilities that they may provide, or constraints that they may impose, and a survey of existing work from which new vocabularies may be derived. They've also provided an RDF schema defining the allowable combinations of vocabulary terms and values in a CC/PP profile.

All simple CC/PP attribute values are represented as literal text values (in XML elements or XML attributes, according to the rules for RDF literal object values). In addition to the simple values, a CC/PP feature may have a complex value expressed in the form of a resource with its own collection of RDF properties and associated values. Specific data types represented in this are set of values. Fundamental to the design of CC/PP is the idea that new client attributes can be defined, as needed, through the introduction of new vocabularies. Similarly, new relationships can be introduced through new vocabulary items, though the introduction of these needs a great deal of care to ensure their semantics is adequately and consistently defined. A general principle is that application-neutral CC/PP processors should be able to understand and manipulate CC/PP relationships without necessarily understanding the CC/PP attribute names to which they refer. New vocabularies are introduced through XML namespaces. Their relationship to other CC/PP vocabulary items can be defined by new RDF schema.

3.5 Quality of Service (QoS) Using Congestion Management

The quality of service is the main concern in contemporary wireless networking environment. Good Quality of service consists of many factors, but the most important ones are low delays, low packet loss rates and large bandwidth (high throughput). These factors are in many cases intertwined. Because bandwidth of wireless networks are in most cases narrow, the buffers easily overflow and this causes delays and packet losses. Radio connections are also error prone which causes more packet losses and more delays, because the lost packets have to be retransmitted. This again wastes the narrow bandwidth and causes more troubles.

The buffer overflows of slow wireless connections are probably the biggest problem in the wireless agent communication. The situation, in which user is watching a video stream over slow connection, is likely to reserve all transmit buffers. When the large stream occupies all resources there is not any more space for even control traffic. If user wants to stop the video stream it might take long time
before the control message to sender to stop sending gets through. (If there are proxies along the route it might take a while until the possible buffers of proxies are empty.) This kind of situation is called congestion and the solution is to have traffic management with different traffic classes.

There are at the moment many proposed solutions for Internet congestion avoidance and QoS control. Integrated Services (IntServ), Differentiated Services (DiffServ), Multiprotocol Label Switching (MPLS) and policy based networking such as the Common Open Policy Service (COPS) are just some examples of these. None of them are in widespread use in the Internet yet. Most promising for CRUMPET seems to be DiffServ [IET00a]. It uses the Resource Reservation Protocol (RSVP) to reserve resources for data flows. DiffServ can reserve some fixed amount of bandwidth for control traffic to hinder control message jams.

In the case of CRUMPET just traffic between Access Node and wireless terminal need to be controlled, since this is supposed to be the bottleneck. It might be impossible to make resource reservations along the entire path to content providers since the CRUMPET content services might reside anywhere in the Internet.

The small footprint agent platform needs to have the capability to somehow show to the congestion management service, which of the messages are important control messages and have to be passed through in any circumstances. The simplest solution would be to tie certain TCP or UDP ports to congestion management service and declare all traffic through these certain ports to be control traffic. More elaborated solution would be an API in agent platform to send messages which belong to certain QoS group.

The actual implementation of congestion management is not in the scope of this deliverable, but it is here taken into account to ease the implementation in later phases.

3.6 Summary

If the client side computer is a small device with limited computing power, we need to refit some of the agents residing on the mobile node. The agents need to be small; they should not use too much computing power. In addition, the communication capability of the device at various times is subject to change, so we cannot expect that the communication link is always open.

In essence, the agents intended for the small device have to take into account three factors:

- Limited memory (RAM, ROM, persistent storage)
- Limited processing power
- Limited connectivity

Depending on the execution environments available on the small device, these agents can be implemented using various software engineering techniques presented in this section:

- Standard 1.2 Java AE
- PersonalJava standard
- Java Micro Edition, KVM
- Proprietary C or assembly language implementation.

If the small device is a laptop, we can use a full-blown agent platform such as the FIPA-OS and the standard Java platform. However, if the device is more limited, we can use custom components tailored for the device, or port the agent platform features that are most critical. The following table illustrates the programming environment possibilities regarding the design and implementation of the FIPA-OS agent platform for constrained environments.
Currently there are two options for Java developers interested in creating software for small devices. Both the PersonalJava and the CLDC (KVM) examined in section 3.1 may be used to create small footprint Java applications. However, since the CRUMPET services require the FIPA-OS platform or part of it and are planned for a high-end PDA type of device, the CLDC is not suitable for such an environment. CLDC targets the low-end PDAs and mobile phones, which effect stronger limitations. Since the CDC configuration that is part of the Java 2 Micro Edition will incorporate the PersonalJava as a profile, software written for the PersonalJava programming environment will be supported in the future.

Appendix A summarizes the Java support and features of different small devices. The CRUMPET services target the high-end PDA and currently the Compaq iPAQ, based on the 206MHz StrongARM processor, is the fastest PDA on the market. The iPAQ may reflect best the capabilities of the future medium level PDA. One of the key issues with the iPAQ is its extensibility. The device supports PCMCIA type II cards and offers the possibility of using GPS at the same time with WLAN, for example. Appendix E summarizes the hardware features of the iPAQ.

The iPAQ is shipped with the PocketPC operating system from Microsoft. A PersonalJava development environment and virtual machine can be downloaded from the Sun web site for PocketPC and the StrongARM (iPAQ) and MIPS processors (Cassiopeia). In addition, the open source Linux operating system can be used with iPAQ and also Cassiopeia. In order to install Linux, the existing PocketPC operating system is removed and possibly saved for a later restoration. Afterwards, the Linux Kernel, libraries, and applications are transferred to the device using a standard cable connection. After the Linux system is functioning, new software may be transferred using a cable (for example using a PPP connection). Linux running on the PDA has the same features as any Linux configuration: the X-windowing environment (X11) may be used and the same software may also be used, after they have been recompiled for the processor. The PersonalJava development environment may also be used on the small footprint Linux with the open source Kaffe virtual machine. In addition, the PocketLinux system may also be used which uses Java to create a PDA style interface for the device.
The following table summarizes the operating system and Java platform possibilities for the CRUMPET project. The current design is based on the Linux operating system tuned for PDAs and the open source Kaffe virtual machine from Transvirtual. Since the PersonalJava code is very portable, adhering to the PersonalJava standard does not prevent from running the software on the PJava implementation from Sun, for example. However, device driver interfaces created using JNI will need to be adapted if the underlying operating system changes, for example from Linux to PocketPC.

<table>
<thead>
<tr>
<th>Feature</th>
<th>PersonalJava</th>
<th>SUN</th>
<th>Kaffe, PocketLinux</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI (Remote Method Invocation)</td>
<td>Yes</td>
<td>Partial, under development.</td>
<td></td>
<td>Might be needed for Jini (lookup service)</td>
</tr>
<tr>
<td>JNI (Java Native Interface)</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>Important because we need to use native code in order to access device hardware and OS.</td>
</tr>
<tr>
<td>AWT</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>Needed for the User Interface.</td>
</tr>
<tr>
<td>Support for MP3, Flash, MPEG. (Not part of the PJava)</td>
<td>No, may be available as third party programs.</td>
<td>Yes for PocketLinux.</td>
<td></td>
<td>PocketLinux includes support for media. Standard PJava does not include these.</td>
</tr>
<tr>
<td>Java.reflect and java.beans packages</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>Reflect is needed by FIPA-OS</td>
</tr>
<tr>
<td>XML</td>
<td>No, third party</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devices</td>
<td>MIPS,StrongARM, SH3, SH4</td>
<td>MIPS,StrongARM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>Windows CE 2.11 or newer, PocketPC</td>
<td>Linux, Kernel 2.4.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>1.1 Beta 1</td>
<td>1.0.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PersonalJava Specification</td>
<td>1.1.3</td>
<td>1.1.1</td>
<td></td>
<td>The 1.1.3 specification introduced some clarifications and corrected a few errors. Otherwise the functionality is the same.</td>
</tr>
<tr>
<td>Source code available (OS, Java)</td>
<td>No</td>
<td>Yes, both.</td>
<td>Source is important, because we may need to introduce changes in the communication protocols.</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>HTTP</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDL CORBA</td>
<td>No</td>
<td>No</td>
<td>Corba ORB support is available as third party products, such as HORB [HOR00].</td>
<td></td>
</tr>
</tbody>
</table>
4. MicroFIPA-OS

This section presents and evaluates the design of a small-footprint version of FIPA-OS. Section 4.1 presents the design overview, section 4.2 examines the MicroFIPA-OS components and the Agent Shell which is the core of a FIPA-OS agent. Section 4.3 presents the FIPA compatible agent platform for small devices, and section 4.4 a partitioned platform that distributes some of the platform functionality on the fixed network. Section 4.5 examines software partitioning using profiles and section 4.6 discusses connection failure issues. Finally section 4.7 summarises the whole section.

4.1 Design Overview and Restrictions

There are some limitations to take into account in the design of the small footprint agent platform, because it should work efficiently on a PDA device of small amount of memory and processing power. The bandwidth and quality of connections will vary to a great extent. Sometimes connection maybe GSM-data with less than 10kbits/s throughput and sometimes wireless LAN with throughput of over 1000kbits/s. With wireless connections the problems considering quality of service are relevant. There may be lots of errors in transmissions, which the lower levels of transportation services have to cope with. It may happen that connection suddenly vanishes. The agent system has to be prepared to situations like this.

In the CRUMPET project it has been agreed that design should follow the FIPA (Foundation for Intelligent Physical Agents) standards. It has also agreed to some point that the platform is based on FIPA-OS agent platform originally developed by Nortel Networks. This platform follows the FIPA specifications as well as possible. FIPA specifications themselves do not give exact guidelines how to implement a real platform. The FIPA abstract architecture specification [FIP00] defines the minimum mechanisms that are required for an agent platform to be FIPA compliant. These are agent registration, agent discovery and inter-agent message transfer.

The FIPA specifications are very abstract and define agents and the agent platform at a very high level and they do not restrict the actual design much. What really restricts the design is how closely we follow the current design of FIPA-OS. It is clear that some modifications are needed, since the FIPA-OS performance using the PersonalJava virtual machine on a PDA device was observed to have very high response times.

The agent platform in the PDA device should offer the programmers and agents an environment as close to the normal agent environment on standard workstations. If we try to modify the current FIPA-OS to work on PDA-devices, it is reasonable to keep the API as close to original as possible. According to the FIPA-OS developers guide [NOR00a] the components, which offer the API for an agent programmer, are the agent shell, task manager and conversation manager. It should be noted, that the agent shell is not needed when one implements an agent in FIPA-OS [PBH00,PBH00a].

Improved performance of FIPA-OS on a PDA device can be achieved in many ways. The current hot spots can be identified, analyzed and removed. Some of the functions can be moved to the fixed network side. DF (Directory Facilitator) and AMS (Agent Management System) are the key components and agents of any FIPA compatible platform. The relocation of functionality can be taken so far that PDA acts only a user interface. In this case almost all of the work is done on the network side and the communication between the PDA and the fixed network side of the platform is mainly transmitting outputs to user and inputs from user. The agents then reside on the fixed network side. When there is a large number of mobile users using one fixed network agent platform part, there appears scalability problems. FIPA-OS agents use tasks, which are run as Java threads. One agent may have dozens of tasks and if there is many of these agents by many users the scalability is a problem, which is not easy to overcome.
A more reasonable way to implement the agent platform for PDA devices is to put some work on the network side, and most of the work to the terminal side. To go further this way, whole agent platform can be implemented on PDA-device.

4.2 MicroFIPA-OS Components

This section introduces the internals of the MicroFIPA-OS and the different ways the agent platform functionality can be supported on the mobile device. We begin with an overview of the agent shell and present the different components of the shell and how they can be optimized for constrained environments.

4.2.1 Overview

The MicroFIPA-OS consists of a hierarchy of components, which are illustrated in figure 10. These components are MTPs, MTS, profiles, the conversation and task manager, the agent shell and possibly the mandatory FIPA platform agents.

![MicroFIPA-OS component overview](image)

In order to support FIPA standards we need also to have an ACC if the MicroFIPA-OS is a full-fledged FIPA platform.
4.2.2 FUNCTIONALITY OF THE AGENT SHELL

In FIPA-OS the Agent shell is implemented with the FIPAOSAgent class that is part of the fipaos.agent package. The Agent shell does not specify the internal structure of agents, but the services that agents may use in order to communicate with the agent community and to use the local infrastructure. The Agent shell and also agents are characterized by profiles in the agent platform. The Agent shell can have various capabilities depending on the execution environment.

Agent shell requirements:

- MTS: Message Transport Service, message queue, incoming and outgoing messages. Interface to the agent community through ACC. ACC needs to be loaded before the Agent shell can receive or send messages to other systems.

- Parsing packages. ACL parsing, SL0 (required by FIPA Agent Management Ontology) parsing, optionally XML or RDF parsing.

- Profiles that define what transport options are available.

- Tasks, which allow the agents functionality to be decomposed into “reactive” sub-behaviours according to the agent paradigm. Different tasks call for task management.

- Conversations, which handle various interaction protocols specified by FIPA. Conversations automate the process of receiving and sending messages pertaining to different phases of the interaction protocols.

The Agent Shell in FIPA-OS provides the following functionality:

- Sending messages using the forward() – method. Messages may be sent in either the FIPAOSAgent or Task class.

- Retrieval of the agents’ properties such as Profiles, AID and state. This includes locating the platform agents DF and AMS.

- Registration with the platform agents: registerWithAMS() and registerWithDF(). This functionality is provided in the form of tasks: DFRegistrationTask and AMSRegistrationTask. Using these behaviours results in callbacks to the FIPAOSAgent-class.

- Setting up tasks and access to TaskManager.

- Shutting down the agent and its components.

Possible optimizations:

- The parsing of messages creates overhead, and should be avoided for example by using method calls, which are faster and more efficient. Agents that use message passing as the only means of communication need constant message parsing, marshalling and unmarshalling. Access to Agent Management Service (AMS) should be preferably used via method invocations in one platform, not using ACL messages with SL0 content as specified by FIPA. However, some parsing is needed, because it would be very limiting from the FIPA viewpoint to remove ACL and parsing altogether.

- Task paradigm can be replaced with the Java threads, with the cost of having to implement a new task management system or re-implement the existing one. Task manager gives the
agents very basic functionality like easy message sending and receiving, and registering with the DF and the AMS. This optimization would make agents seem more like traditional multi-threaded objects, while simplifying the agent management code. An additional non-preemptive or preemptive task management level creates overhead. Another optimization issue is to minimize or restrict the number of concurrent tasks.

In FIPA-OS the agent shell functionality is realized in the AMS, DF agents and the ACC. In FIPA-OS both AMS and DF are specific types of agents. ACC is a lower-level entity and part of the MTS (Message Transport Service). FIPA-OS supports both CORBA naming service and RMI naming service. The Agent Loader cannot load the ACC, so it must be started separately. An Agent Loader is used to load arbitrary agents into the same VM. One VM is used by each instance of the Agent Loader. Agent Loader can also be used to shutdown agents via a GUI. Agents that need to be used by the Agent Loader have to implement the constructor:

```java
public MyAgent(String platform_profile_location, String name, String owner)
```

All agents in the CRUMPET project will be built by extending the agent shell as it allows agents to be built from a small set of components as illustrated in figure 11. The lower three components are fundamental to FIPA compliant agents; these deal with the messaging construction/parsing. The upper three components categorise the agent, its domain, its goals, and any special abilities it might have (e.g. negotiation, mediation, brokerage, etc.). Additional components will be constructed to implement functionality such as agents to handle geodata [CRU00].

![Figure 11 The Agent Shell.](image)

### 4.2.3 TaskManager

The standard TaskManager component in FIPA-OS enables multiple conversations and also other activities to be conducted simultaneously. The Task API provides also method to search the DF- the searchDF method that uses the DFTask – class.

Since Tasks are implemented with Java threads and a certain overhead is involved in the creation and deletion of these threads, we can limit the number of simultaneous conversations per agent. The number of active threads should be also specified in profiles. It is also possible to simplify task management by limiting the number of active tasks and re-implementing often used tasks, such as the
DFTask which queries the Directory Facilitator, to be more efficient. In addition, any optimizations made to the conversation manager reflect also to the management of tasks, because tasks use conversations heavily.

### 4.2.4 Conversation Manager

ConversationManager groups messages of the same conversation together. Conversation represents individual conversation and keeps track of the FIPA interaction protocol state.

ConversationManager implements the MessageReceiver interface, which allows it to deal with incoming messages and the MessageSender interface which, enables other components to send message via it. In addition, ConversationManager has a reference to MessageSender implementation that allows it to send messages. Furthermore, it has a reference to ConversationListener to allow it to pass Conversations to components that implement this interface.

Various Conversation classes implement various FIPA specified interaction protocols.

**MicroConversationManager**

Since it is probable that the agent-based application running on the MicroFIPA-OS only requires a subset of FIPA specified interaction protocols, we may have several implementations of the ConversationManager for different needs. There is always one implementation of the ConversationManager available and running and it may be the default version with full features or an optimized version with a limited set of features. For instance, fipa-request is needed to access various platform services, but it is doubtful whether we need to have support for different auction protocols. This gives the motivation for pluggable ConversationManagers for agents that do not need the whole set of protocols, but use for example one-shot interactions. The subset of supported interaction protocols is defined in the MicroFIPA-OS platform profile. In addition, the supported pluggable ConversationManagers should be also available in the profiles. Now, the agent platform determines the best-suited ConversationManager for the agent-based applications at start-up; agents define the required interaction protocols in agent profiles.

Furthermore, since conversations are implemented with Tasks (Java threads) and a certain overhead is involved in the creation and deletion of these threads, we can limit the number of conversations to a fixed number of simultaneous conversations per agent. The number of maximum active tasks should be also specified in profiles. This may be implemented at the TaskManager level, where the number of threads is limited by thread pooling. Now, we are limiting the number of parallel threads, but not the flexibility of the system.

Standard FIPA-OS conversations can be stored to disk for later retrieval and analysis. This feature is not needed on small devices and should be optional if needed at all. Information whether conversations are logged or not should be also kept in profiles.

### 4.3 Full Functional Agent Platform

If the agent platform for small devices consists of two dependent parts, namely terminal part of platform and fixed network part, the terminal is dependent on the network part of the platform. There always has to be the network side of the platform in this design (Access Node), otherwise platform is useless. This limits the interoperability with other FIPA compliant agent platforms like JADE or ZEUS [Bte00]. To overcome this problem we can implement a full functional FIPA agent platform for mobile devices.
The overall design of the CRUMPET platform in the case of full functional platform is presented in figure 12. First the Mobile Platform contacts to its so-called home platform to get in touch with some local platform. The contact can be done using ACL messaging or it can be implemented at lower level, even by using plain sockets.

The home access platform is located using information stored in the platform profile. However, there are more complicated methods of doing the home platform discovery, but the first implementation will have a profiles based solution. The next paragraphs present discussion about more sophisticated platform discovery protocols.

One convenient way to do the platform discovery query is to use the Director Facilitator agent available on the home platform or some well-known platform. In this case connection is done between agents using ACL messaging. The platform returns then the address to the nearby platform, which offers the CRUMPET services.

If the DF of well-known platform is used for platform discovery, it should somehow know about (all) the available CRUMPET platforms. Simple solution is to have just one well-known root platform and DF there to tell the platform addresses. Every CRUMPET platform would then contact this platform to advertise its services. This design is not very scalable and fault tolerant, because there is only one well-known platform for service discovery. FIPA does not offer any solution to this problem. Some kind of DNS (Domain Name System) style federation and hierarchy of DFs would be a good solution [CDK94]. Since in the CRUMPET trial we more likely do not encounter scalability problems, a feasible solution might be to have two DFs with well-known addresses. Each CRUMPET platform registers itself to both of these. If a mobile agent platform cannot access one it can still try the other one. There are more complex and fault tolerant solutions for service discovery presented in literature [CDK94], but their benefits compared to the approach presented here might remain quite negligible.

Communication between other agent platforms happens using ACL messaging. It is important to notice that now access nodes in figure 12 can in theory be any FIPA compliant platforms, which can interoperate. Architecture is simpler with clear interface to fixed network.

Figure 12 The overall architecture of CRUMPET system in full agent platform design.
The design of full functional agent platform for mobile devices can follow closely the design of FIPA-OS. The performance and memory critical parts of the software need to be simplified and redesigned. Major components of current FIPA-OS can be implemented in order to keep the API same for agent programmer. Conversations and tasks are the most important components of FIPA-OS to application developer.

Tasks and conversations in FIPA-OS use the Message Transport Service. Because MTS is not used directly by an agent programmer, the underlying implementation can be changed without changing the interfaces and thus this does not affect the agent programmers. It is possible to gain performance by simplifying the current implementation of MTS. The main components of the MTS in FIPA-OS are shown in figure 14.

The FIPA-OS MTS is based on stacks and uses an internal and external service stack for messages. Each stack consists of a number of services that are applied to the messages. Each service is a component that performs some operation on the messages passing it. This layered architecture is adaptable for a wide range of message transport services and changing a transport involves changing a layer within the desired message stack. As figure 14 illustrates the current implementation of MTS consists of several classes, abstract classes and interfaces. This structure can be adapted for constrained environments by simplifying the message stacks. This is possible, because the agent shell and other components, such as the conversation manager, that rely on MTS use the transport features through abstract interfaces.
4.4 Partitioned Platform

One of the central design issues with MicroFIPA-OS is the distribution of the standard FIPA-OS agent platform into a number of mobile hosts. These mobile clients would support a subset of the standard agent platform and distribute functionality that cannot be supported on the mobile node to a server computer running on the network. The actual host computer of the FIPA-OS that hosts the AMS, ACC and DF is referred to as the network entry point or the CRUMPET Access Node. The mobile nodes and agents residing on these devices connect to the network entry point in order to send and receive messages and use other agent services, such as tourism based information services. An alternative design is presented in section 4.3 where, instead of dividing agents within the platform into a number of hosts, the mobile device hosts a full FIPA compatible agent platform and agents residing on this platform use ACC to access CRUMPET service platforms.

With a partitioned platform we have several levels of distribution: first we have the micro platforms which are part of the FIPA-OS platform, then we have the platform internals mandated by FIPA, AMS, ACC and DF, which may also be distributed. In the JADE agent platform [CSE00] they are always on the same host, although the platform may consist of several hosts (each host hosting a container in JADE terminology). This “agent-container” approach is simpler than a scenario where AMS, DF and ACC can be deployed on different hosts. From the small devices viewpoint the location of AMS, ACC and DF does not matter if they can be reached efficiently from the small device. If the
agent uses ACL and the system is capable of forwarding messages within the domain, the location of the AMS, ACC and DF becomes a configuration issue.

The main design issue in small mobile device platform is to offer as much as services as possible to agent application programmer, but still maintaining the performance of the agent platform. In this sense it is reasonable to offer some services locally while some services are executed on the fixed network side with more powerful computer. For example agents could create ACL messages locally while the ACC, which is responsible for routing messages to agents on other platforms, would reside on the fixed network side.

Figure 15 illustrates a scenario where many mobile small device agent platforms called Micro Agent Platforms (uAP) are connected to one fixed network side agent platform called Fixed Network Agent Platform (fAP). It should be noticed that uAP (or many uAPs) and one fAP create one FIPA agent platform.

Bootstrapping sequence of the uAP is shown in figure 16. When uAP is started it reads its local profile. In that profile there is saved the Home Agent Platform (hAP) address of uAP. Home Agent Platform acts here in the role of a platform broker. It maps for the uAP the best suited local platform. The mapping is done according to the information, which the Home Agent Platform gets from the Micro Agent Platform. This information might contain geographical coordinates from the GPS device or input from the human user, where he or she is now. In some cases just the current IP address of the uAP might be enough to guess where the user is because sometimes traceroute and nslookup facilities of IP reveal the user location. (In mobile connections, when the Point to Point Protocol is used this method is not feasible.)

After the uAP has got the address of local fAP it tries to register to it. The fAP either accepts or rejects the registration according to its security restrictions. Since security issues are not the main research areas of the CRUMPET project, they are now just taken into account in this design so that they are easy to implement later. After registration is succeeded uAP asks for the platform profile of fAP. Among this profile it possible for uAP to dynamically configure its setup. It is sensible to be able to dynamically adapt to different kind of fAPs. It might be the case that sometimes fAP offers plenty of services to uAPs and sometimes there might be just minimal set of them. Also the computing environment (physical device, operating system, middleware, available memory), where uAP is executed may change and it is rational to adapt to different environments by being able to change configuration.
Agents in CRUMPET communicate by using FIPA ACL messages. In the FIPA architecture it is left to the problem of FIPA specified entities: Message Transport System (MTS), Directory Facilitator (DF) and Agent Management System (AMS) to transfer the agent messages to the right agents [FIP00b]. FIPA does not specify the internal implementation of these entities but it is left for the responsibility of the platform programmer.

The uAP resides on mobile device, which means that sometimes the bandwidth of the link might be very narrow and the round trip times long. Because of that, the number of requests over wireless link should be kept as small as possible.

4.4.1 MESSAGE TRANSPORT SERVICE (MTS) IN A DISTRIBUTED FIPA-OS

In FIPA-OS the MTS provides the ability to send and receive messages. The MTS has two functions: process incoming and outgoing messages. This is realized in an implementation of internal and external service stacks for messages. Each stack consists of a number of services that are applied to the messages. Each service is a component that performs some operation on the messages passing it.

The Internal Message Transport System of CRUMPET platform is responsible of routing the agent messages to the right agents. It does this by keeping an internal database of the uAP addresses. For example the IP address of uAP might change when horizontal handoff is made. One example of this kind of situation is when WLAN connection is changed to the slower GPRS connection. This involves the changing of IP address. In this sense the uAPs need more static addresses. There is two basic solutions, either the platform has one globally unique address or then it just has temporary address allocated by fAP for one session, which might then in some cases last even several days. The problem with globally unique addresses is that there must be some authoring the use of addresses. In the trial of CRUMPET this definitely is not a problem, but in real life global distributed systems like Internet it really is.
On the MicroFIPA-OS system we need only to have an internal message transport and possibly an internal message queue for intra-device messages. The IMTS protocols may be proprietary and may involve the use of the Access Node (figure 1.I). However, on the FIPA-OS domain we need to have the capability to distribute agents to a number of mobile hosts.

This design requires that the MTS can route messages within the FIPA-OS domain to mobile terminals running the MicroFIPA-OS system. This internal message routing can be facilitated by having an internal router service placed in the MTS service stack (figure 17). This MicroFIPA-OS Message Router component would be needed in both external and internal MTS service stacks. The MicroFIPA-OS may need also a message buffering service in order to store messages for later delivery. This is needed because small mobile devices have limited connectivity. The Access Node or a proxy service may also provide message-buffering functionality for the MicroFIPA-OS systems. In the CRUMPET architecture the Access Node may support the buffering of messages.

**MicroFIPA-OS Message Router**

The MicroFIPA-OS service component routes outgoing messages to MicroFIPA-OS systems through the Access Node. The Router component does not necessarily need to process incoming messages from MicroFIPA-OS systems if the Access Node is part of the FIPA-OS platform and forwards the ACL messages down the appropriate message stack. However, the Router component is needed on both internal and external MTS service stacks.

---

**Figure 17** FIPA-OS Message Transport Service (MTS) logical composition.
Message Buffering

Message Buffering should be done as near as possible to the MicroFIPA-OS Message Router. The message buffering should be activated if the MicroFIPA-OS Message Router is incapable of sending the message to the mobile terminal, which might be temporarily unreachable. The message buffering service keeps the messages for mobile terminals a given time (keep-time) and tries to forward the message again. If a predetermined number of retries fail or a given time constraint is satisfied, MTS cannot send the message and it is classified as undeliverable. The message buffering capability will be the feature of the CRUMPET Access Node ACC.

4.4.2 MICRO AGENT NAMING IN A DISTRIBUTED AGENT PLATFORM

Provided that the agent platform is distributed the agent discovery has to be solved somehow. There might be agents having the same name residing on different micro platforms. Agents should have unique names in the domain of each distributed platform. Since every platform has unique name those names are unique in the domain of the whole agent universe. There are several possibilities to form unique agent names.

Since the agents residing in the MicroFIPA-OS are part of a larger single FIPA-OS domain, we need to have a naming scheme that allows a large number of agents running on the same agent platform. The FIPA-OS MTS needs, based on the agent name, to know to which MicroFIPA-OS forward the message. Therefore there is a requirement for a systematic naming scheme for these agents.

The addresses in the CRUMPET system are formed in the following syntax: agent1.uAP@fAP. Agent1 is the name of the agent, uAP is the name of the mobile MicroFIPA-OS platform and fAP is the name of the CRUMPET Access Node (the fixed network agent platform). The ACC, which is in response of routing the messages coming beyond the local platform, has a cache of the actual transportation addresses of uAPs. The address personalAgent.CRUMPETPDA@HelsinkiPortal is in the ACC of HelsinkiPortal translated to actual network address with port number like 128.214.9.55:3030 through which the actual agent (personalAgent) can be contacted. ACC then routes the message to that address where the internal message transport system of the uAP (in this example CRUMPETPDA) gets the message. If the uAP cannot be contacted for some reason then ACC buffers the message and tries to send the message again later. There are many reasons for connection breakdowns and they are very likely with wireless connections so the internal message transport systems in both uAP and fAP have to be prepared that connection may be lost at any given moment.

4.4.3 AGENT STARTUP

The agent startup sequence is shown in figure 18. First PAgent, which resides on uAP wants to register itself according to the FIPA specification to the agents management system agent (AMS). To make this it uses the services of Internal Message Transport Service (IMTS). IMTS contains an address resolution cache, which is used to speed up message routing. The IMTS of uAP forwards the message again to the IMTS of fAP, which in turn delivers the message to the AMS. AMS sends back an agree message and registers the PAgent to the AMS. After this it still sends an inform message to inform the PAgent of successful registration to AMS.
The search of an agent address from AMS is basically similar with the registration procedure just shown. Now AMS just returns the requested information about the agent. Registration to DF is also basically the same.

When an agent residing in the uAP wants to send a message to an agent which lies on remote platform it can send the message through Agent Communication Channel (ACC) agent. ACC is responsible of routing the messages to the remote platforms. This service is called the Message Transport Service (MTS) by FIPA [FIP00c]. It is implementation dependent whether there is just ACC, which deals with all the message routing, or if there are separate MTS and ACC. Since there is internal message transport service it is sensible to reuse some of those services in communication between different platforms. Because CRUMPET will have a distributed platform some of those services actually suite quite well to the Internet environment like message buffering.

The sequence of contacting remote platforms works so that the sending agent sends the message first to the ACC through internal message transport service. ACC uses the external message transport service, which wraps the message to a protocol, which the remote platform understands. This protocol might be hypertext transfer protocol (HTTP), Java Remote Method Invocation (RMI) possibly over Internet Inter ORB Protocol (IIOP) or even Wireless Application Protocol (WAP) [OMG00, WAP00]. Protocol might be also something else, which is understood by both the sender and the receiver. The Message Transport Service of receiver gets the message and routes it to the right agent.

4.5 Software Partitioning and Profiles

Software partitioning can be used to divide work and tasks between different nodes of a distributed system according to issues such as server computing capability and bandwidth [KRa00]. Work has been done to ease the burden on, for example, WWW-servers by transferring some computation tasks to the client computers [AYa98]. Today’s WWW infrastructure accommodates this by using the Java applet technology. Users can download applets from the network. However, we face also quite a different scenario with small devices, where we are interested in transferring tasks that are computationally laborious to the network. Now, our problems are finding suitable services on the networks that do the computations on our behalf, and to properly instruct those services (figure 19).
Figure 19 Dividing work between the host and the network. The decision on what to execute and where is influenced by information such as CPU, network latency and bandwidth and software requirements of the desired component or service.

Examples of tasks which may be executed on the fixed network:

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image generation, visualization</td>
<td>An image or menu is created from the given data and sent as a reply.</td>
</tr>
<tr>
<td>Web search</td>
<td>Data is gathered and only data that matches the given criteria is sent as a reply.</td>
</tr>
<tr>
<td>Route plotting</td>
<td>Server generates the response and returns it in a suitable form for the client (text, text and graphics, etc.)</td>
</tr>
</tbody>
</table>

4.5.1 PARTIALLY DYNAMIC PARTITIONING OF AGENT ROLES

An agent may have one or more roles, which depict different activities the agent may be engaged with. Examples of agent’s roles might be monitoring the QoS, maintaining an active map display and receiving information about the activities and events on the route. Partially dynamic partition would allow, at client system startup, to determine which roles would execute at which hosts on the visible network. QoS monitoring and controlling are aspects that need to be present on the device. Roles pertaining to collecting, refining and adapting information may be placed on some other network node. The partitioned software modules may be agents’ roles as described in this paragraph, or tasks or full agents. The unit of partitioning is not as important as the function of the unit and whether it is reasonable to execute it locally or externally.

4.5.2 PARTITIONING USING PROFILES

Partially dynamic partitioning needs information about the current network, security, and device characteristics and link characteristics. QoS monitoring may provide additional information to the mechanism that makes the partitioning decisions. It is envisaged that the profiles contain at least information about:

- Device configuration.
- Personal preferences.
• Agent configuration that includes the transport addresses of agents and whether they need to be local or can be used remotely.

• Local agent platform configuration.

The most simple way of doing partially dynamic partitioning is to have the local profiles contain both information about the local services and also, if available, information about other nodes on the network that can perform the same service. In essence, the profiles contain sufficient redundancy to make the system adaptable for different constrained environments without requiring mobile objects – the services need to be already installed. For example: we might have an agent that locates restaurants that are located in the vicinity and draws them on a map. If we have a constrained environment the profiles direct the queries to the network node that provides the same service and returns an image that contains the restaurants (in this case we need to have a stub that knows how to display that image and communicate with the external agent). The key benefits of profiles, and especially distributed profiles is that they enable a very modular and dynamic service architecture.

4.5.3 Description for Partially Dynamic Partitioning

The current FIPA-OS profiles can be refined to include support for CC/PP type information and also include information how to execute different CRUMPET services independent of their location. This kind of information may also be available on a DF. CC/PP and FIPA-OS was also discussed in section 3.4.3.

The external configuration information may be local or it may be a reference to a profile that is stored in a remote location. The CC/PP allows the distribution of profile information between clients and servers. The external configuration applies to the current FIPA-OS platform profile and agent loader profile. The FIPA-OS system profile contains information about the FIPA system agents, AMS and DF, whereas the agent loader profile contains information about the user defined agents available on the platform.

4.5.4 Partitioning ACC, AMS, DF and Service Agents

It is possible to partition the agent platform into multiple hosts with the help of the agent platform profile. The ACC, AMS and DF may be used externally by using ACL messages, FIPA Agent Management Ontology with SL0 content and by using a suitable FIPA specified transport that is supported by the FIPA-OS platform. There are possible bottlenecks in scalability even when the AMS, ACC and DF are running on different hosts. However, we may have several ACCs, AMSs and DFs in one domain. Therefore scalability problems can be averted distributing some part of the agent platform activity to new hosts running backup services. In addition, FIPA agents can use whatever means necessary to contact other agents, so the bottleneck of having to do a lot of queries to a single host can be averted.

The partitioning of FIPA system agents, which is a fundamental decision that the platform needs to make, is the most critical. The partitioning of CRUMPET service agents is also possible and even desirable in some cases, but requires some effort in the planning and implementation of the agents (figure 20). It depends on the nature of the agents whether, for example, a stub component is required on the mobile host. In a situation where we have a central user interface agents and a number of agents discussing with a service, we may partition the agents discussing with services if they know how to reach the user interface agents using some means.
4.5.5 Making the Partitioning Decision

Upon boot-up the MicroFIPA-OS agent platform needs to decide whether to use local AMS, DF and ACC or to use them remotely. This is a fundamental decision, because if the AMS, DF and ACC are present the micro platform is a full-fledged platform from the viewpoint of other FIPA domains. If the AMS, DF and ACC are not present, then the micro platform is dependent on a FIPA-OS domain that has those mandatory FIPA platform components. Therefore, the latter scenario is more proprietary and requires the FIPA-OS platform or an access node that hosts those integral platform services, whereas the first scenario is more independent.

However, even if the mandatory platform components are present in the mobile device, it may still need to register its agents on an access node for the agents to be reachable from other platforms. The CRUMPET Access Node is envisaged to act as an intermediate between the mobile platforms and the CRUMPET service platforms on the fixed network.

After that fundamental decision, the platform may upon consultation with the profile information, decide for each agent separately whether to execute them in the local platform or initiate a copy in some remote platform or use an instance already running on some platform. Now, depending on the boot-up phase, the agent platform either already knows the AMS of the remote fixed network platform (reference of the platform is obtained from platform discovery procedure) or has to acquire through DF query or from the profiles. In either case the platform may initiate an agent by sending an ACL message to the AMS of the desired platform. AMS is responsible for maintaining the agent life-cycle. It is envisaged that the decision on where the DF, AMS and ACC are located is done only when starting the whole agent platform.
The profiles contain the basic information on what resources the platform may use and where they are located. Additional information can be obtained from the operating system and the QoS monitor agent, which keeps track of the link quality. The operating system and device profile contain information such as total memory, memory available, heap size, display resolution, availability of persistent storage, CPU type and speed, input devices, speed of link etc. The platform profile gives information about the available interaction protocols, maximum number of concurrent agents and the maximum number of tasks.

For each agent that may be partitioned the profiles contain information on how the partitioning can be accomplished. The properties of the wireless link are important and the following issues may be specified in the profile [FIP00a]:

- line-rate, bandwidth in one direction.
- throughput, the number of user data bits that have been successfully transferred.
- rtt, round-trip-time for a data segment, latency of the link.
- mean-up-time, the expected uptime of an established link.
- ber, the ratio of the number of bits errors to the total number of bits transmitted correctly over a link.

The FIPA Nomadic Application Support specification [FIP00a] considers the adaptation of applications to various nomadic computing environments. Issues that the agents and the agent system needs to consider are: selection of MTP and Message Transport Connection, selection of ACL and content language representation, content adaptation. The FIPA Nomadic Application Support specification defines agents to control and monitor an MTP and the underlying Message Transport Connection (MTC). The specification also defines an ontology, which can be used to represent quality of service of the MTS in the nomadic application support context.

**Interface for Agent Partitioning**

Agents that want to give extra information about partitioning may implement the agent partitioning interface. This interface provides similar sort of information as the profiles, but the agents themselves may reflect changes to the information on the fly giving more possibilities for dynamic partitioning. Using this interface, the system may gather knowledge on the fly about agents’ requirements and current status and possibly repartition the system structure based on this information.

4.5.6 **DOWNLOADING CODE FROM THE NETWORK**

From the constrained device viewpoint we have two rather different requirements: one is to keep the system efficient by using resources that are best used on some other node in the network, and on the other hand we may need to dynamically download code to the client.

The future Java enabled (KVM for example) may allow the downloading Java classes dynamically from the web in the form of midlets or applets. This has been one motivation for J2ME CLDC and the Mobile Information Device Profile that defines the capabilities of midlets. J2ME and CLDC were discussed in section 3.1.4. As such FIPA-OS does not support agent mobility, and code mobility can be implemented at different levels of the platform. It can be implemented at the platform level, where agents themselves can be mobile or it can be implemented at the agent level, where agents consists of mobile parts such as tasks (Appendix C discusses mobile tasks).
Agent mobility is more difficult to implement in contrast with object mobility, because agents have a state and some active relationship with other parties. When an agent moves, it has to deal with the changes in the transport address, pause ongoing conversations and after transmission of the code, the system must be able to reinstate the agent properly so it can go on its business. Java supports the transmission of instantiated objects with the Serializable-interface. Serializable objects can be written to streams. Moving objects that have no active state is easier, and the objects can be instantiated after they have been transmitted.

Therefore, the downloading of dynamic code and code mobility could be done in the form of Java objects or JAR files. To ease the developer a library that supports transmission of Java objects with CRC checking could be supplied with the MicroFIPA-OS. Agents may also be transferred as Java classes and instantiated when required however in this case they would not retain any active state. It is also possible to add aspects that support agent mobility into FIPA-OS.

4.5.7 CC/PP Profiles

W3C in co-ordination with the WAP (Wireless Application Protocol) Forum is developing the Composite Capabilities / Preference Profiles (CC/PP) specification [WAP00, W3Ca]. The RDF (Resource Description Framework) based CC/PP is aimed as the standard for device and user agent preferences. The profile information would include display, keyboard and user agent information. Next generation servers could use this profile information to create device specific stylesheets and thus be able to serve a wide range of different Internet appliances and devices. In addition, a new or unknown device may be mapped to a profile with the closest set of features.

Example CC/PP profile:

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
    xmlns:ccpp=http://www.w3.org/2000/07/04-ccpp#
    xmlns:uaprof=http://www.wapforum.org/UAPROF/ccppschema-19991014#>
    <rdf:Description about=http://www.ist-crumpet.org/CrumpetDeviceProfile>
        <type resource=http://www.ist-crumpet.org/profiles/schema#hardware />
        <uaprof:CPU>StrongARM</uaprof:CPU>
        <uaprof:ScreenSize>240x320</uaprof:ScreenSize>
    </rdf:Description>
</rdf:RDF>
```
Figure 21 An example CC/PP profile distribution between the mobile node (MicroFIPA-OS) and the fixed network platform.

In this schematic model the interaction between platforms and agents proceeds as follows (figure 21):

1. MicroFIPA-OS platform determines the agents to be executed at start-up (and possibly other agents as well). Information pertaining to agents is retrieved from the profiles.

2. The profile for agent A contains both local and external profile information. The platform determines that the local copy cannot be run, because of missing interaction protocols and memory.

3. The external profile is retrieved from the platform residing on the fixed network. This is not necessary if the local external information is judged to be valid or there are no URIs, but it may be necessary to update the local external information.

4. The component does not place any requirements on network latency or bandwidth so the platform decides to execute the agent on the network. If the agent is already running this is unnecessary. Security issues are not considered here. After the execution of the agent, all requests for the services provided by the agent should be sent to the fixed network platform. If an agent is used externally, it may also be necessary to create a stub on the small device that creates the local display or otherwise bridges the external module with the small agent platform.

In practice this kind of profiles may be difficult to maintain and a similar system can be achieved using the DF as the basic storage of profile information. DF can be used to store and query information about an agent and agents’ services. In order to make the maintenance of the agents easy; a central repository of profile information is needed.

4.5.8 FIPA-OS PROFILE SUPPORT

In addition to partitioning information we need to keep a list of available interaction protocols and tasks in the platform profile. FIPA does not mandate to use any particular communicative acts or interaction protocols in agent to agent communication, although SL0 and the Agent Management Ontology have to be supported by the ACC, DF and AMS. Also, knowledge of available content language parsers is needed. The platform may also restrict the number of concurrent tasks and agents. An agent or the Agent Platform must, at start-up, determine what is available and what cannot be
provided that the agent requires (figure 22). It may be that the platform cannot execute some agents, because they use protocols that are unavailable.

The FIPA-OS platform profile defines the AMS internal transport address. This and the internal address of the ACC can be hardcoded with this profile. The number of parallel threads can be limited in the TaskManager. In addition, the supported protocols and number of simultaneous tasks, and threads and agents are determined in this profile. The platform must support fipa-request at minimum, because AMS and DF communicate using this protocol. On the other hand, instead of supported interaction protocols the platform profile may define a MicroConversationManager, which is to be used instead of the standard component. Pluggable ConversationManagers may be optimized for specific applications, such as applications using only fipa-request and one-shot interactions.

Figure 22 Overview of the platform startup procedure.

```xml
<!-- FIPA-OS specific details -->
<ap:AMSAddress></ap:AMSAddress>
<ap:ACCAddress></ap:ACCAddress>
<ap:DFAddress></ap:DFAddress>
<ap:SupportedInteractionProtocols>fipa.request</ap:SupportedInteractionProtocols>
<ap:ConversationManager>MicroCM</ap:ConversationManager>
<ap:ConcurrentTasks>7</ap:ConcurrentTasks>
<ap:ConcurrentAgents>7</ap:ConcurrentAgents>
<ap:ConcurrentThreads>7</ap:ConcurrentThreads>
```
FIPA-OS Agent Loader Profile

Agent Loader determines which agents are started at runtime, what is the main class of an agent and who owns an agent. The MicroFIPA-OS extends the FIPA-OS agent loader by introducing more parameters for agent requirements. These can be used in making the partitioning decision. This information may also be available on a DF.

<table>
<thead>
<tr>
<th><strong>Standard Profile tags</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentName</td>
<td>Name of the agent, e.g. PingAgent</td>
</tr>
<tr>
<td>ClassName</td>
<td>Name of the class of the agent e.g. ping.class</td>
</tr>
<tr>
<td>Execution</td>
<td>Local, either or external. Determines the execution options.</td>
</tr>
<tr>
<td>Runtime</td>
<td>Determines whether the agent is started upon platform bootup or not.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Local Profile tags</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Defines the local execution parameters.</td>
</tr>
<tr>
<td>Protocols</td>
<td>The necessary interaction protocols.</td>
</tr>
<tr>
<td>Contentlanguages</td>
<td>The necessary content languages.</td>
</tr>
<tr>
<td>Cpu</td>
<td>Approximate CPU requirements. This can be a benchmarking value or CPU type or some other term denoting a certain performance level.</td>
</tr>
<tr>
<td>Memory</td>
<td>Approximate memory requirements. It is envisaged that this is the approximate memory requirements of the agent in kilobytes.</td>
</tr>
<tr>
<td>Tasks</td>
<td>The number of concurrent tasks needed by this agent.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>External Profile tags</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>External</td>
<td>Defines the external profile.</td>
</tr>
<tr>
<td>Stub</td>
<td>The local stub class file (or agent name) that is needed to interact with this agent.</td>
</tr>
<tr>
<td>Transports</td>
<td>A list of possible transport addresses for this agent. May also be reference to a DF.</td>
</tr>
<tr>
<td>Protocols</td>
<td>Needed protocols. For example fipa-request.</td>
</tr>
<tr>
<td>Ontologies</td>
<td>Ontologies that are needed and supported.</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Contentlanguages</td>
<td>Needed content languages. E.g. SL0, XML.</td>
</tr>
<tr>
<td>Latency</td>
<td>The minimum latency required in order to use this agent.</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>Minimum bandwidth.</td>
</tr>
<tr>
<td>Rtt</td>
<td>Round-trip time (same as latency)</td>
</tr>
<tr>
<td>Throughput</td>
<td>Number of transmitted error free user bits per second.</td>
</tr>
</tbody>
</table>

```
<ap:Agent rdf:about="df">
  <ap:class>fipaos.platform.DirectoryFacilitator</ap:class>
  <ap:agent_name>df</ap:agent_name>
  <ap:owner>fipaos</ap:owner>
  <ap:start>true</ap:start>
  <ap:execute>local | external | either</ap:execute>
  <ap:local>
    <ap:contentlanguages>SL0 XML</ap:contentlanguages>
    <ap:memory>2000</ap:memory>
    <ap:tasks>3</ap:tasks>
    <ap:cpu>high | medium | low | verylow</ap:cpu>
    <ap:protocols>fipa-request</ap:protocols>
  </ap:local>
  <ap:external>
    <ap:stub>proxyAgent</ap:stub>
    <ap:protocols>fipa-request</ap:protocols>
    <ap:contentlanguages>SL0 XML</ap:contentlanguages>
    <ap:encoding>fipa.acl.std</ap:encoding>
    <ap:transports>
      http://portal.com/acc
      iiop://portal.com/acc
    </ap:transports>
    <ap:linerate>1 Kbps</ap:linerate>
    <ap:throughput>2 Kbps</ap:throughput>
    <ap:rtt>2 ms</ap:rtt>
  </ap:external>
</ap:Agent>
```
4.6 Connection Failures

Connection failures happen when the wireless link goes down. When the connection breaks the partitioned platform is more subject to failures and recovery than the full platform, which contains the necessary functionality for stand-alone operation. It is envisaged that the boundary between the fully functional platform and the partial platform is flexible, at least in the case that the Access Node supports the distribution of the platform functionality. The following table illustrates the connection possibilities at startup for both strategies:

<table>
<thead>
<tr>
<th>At startup</th>
<th>Full platform</th>
<th>Partial platform</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connection</strong></td>
<td>Platform is loaded and the CRUMPET Access Node is contacted, services are</td>
<td>Platform is loaded, profile based resources are loaded, the CRUMPET Access Node</td>
</tr>
<tr>
<td></td>
<td>registered and located.</td>
<td>(that runs AMS, DF and ACC) is contacted and the system is registered.</td>
</tr>
<tr>
<td><strong>No connection</strong></td>
<td>Agents are loaded and registered on the local platform as usual. Remote DF</td>
<td>The system detects that there is no connection and loads the AMS, DF and ACC and</td>
</tr>
<tr>
<td></td>
<td>registration fails and must be done later. Internal message passing is active,</td>
<td>activates the full platform mode.</td>
</tr>
<tr>
<td></td>
<td>but outgoing messages either bounce or are stored for later delivery.</td>
<td></td>
</tr>
</tbody>
</table>

The agents are informed of the link status by introducing two new call-back methods:

```java
public void connectionUp() {
}
```

```java
public void connectionDown() {
}
```

The method `connectionUp` is called when the link has previously been down and the agent is waiting for (and requires) connection. The `connectionDown` method is called when the link has been down and has become alive again.
In addition, the connection may break down in the middle of the operation, which is discussed, in the following table:

<table>
<thead>
<tr>
<th>During operation</th>
<th>Full platform</th>
<th>Partial platform</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connection breaks</strong></td>
<td>The ACC of the mobile platform buffers the messages for later delivery if a buffer space is present, otherwise the messages bounce and the sending agent gets an error message. The CRUMPET Access Node buffers messages destined for the mobile node.</td>
<td>Agent management operations are halted for the duration of the pause. (Some operations may be performed locally and synchronized later with the Access Node). Internal message passing functions normally. The Access Node buffers messages. Agents are informed about the break in the connection using a callback method.</td>
</tr>
<tr>
<td><strong>Connection is re-established</strong></td>
<td>The ACC notices that messages may be sent to the correct destination, the CRUMPET Access Node, and forwards messages on the queue and receives pending messages.</td>
<td>AMS interface is re-established and the agent management functionality is reactivated. Messages from the Access Node are processed and agents are informed about the re-established connection using a callback method.</td>
</tr>
</tbody>
</table>

### 4.7 Summary

This section has presented and examined the MicroFIPA-OS platform that is either a full FIPA compatible agent platform or part of a FIPA domain through some protocol that may or may not be proprietary. The role of profiles was emphasised in making the decision on what kind of system configuration is needed. One part of the optimization of FIPA-OS is the configuration; what components are available locally and what are available on the network. This makes a difference because local components offer fast response times but also use the local memory space and consume local processing power. Distributed components place the burden on the communication link, and even if a part of the configuration is executing on a remote environment, we still need a component that initiates the request and processes the response. The following table illustrates the benefits and disadvantages in partitioning the platform.
From the viewpoint of the CRUMPET services, we are interested in the efficiency of the system; both communication and performance wise. It seems that in order to improve the agent system performance, we have to provide some sort of partitioning of the agent platform functionality and combine the best features of the different designs. The system may be able to function also as a full FIPA compatible platform, but it also works in connection with the CRUMPET Access Node, which hosts some of the mandatory platform components. Therefore the system consisting of the mobile clients and the Access Node becomes a FIPA domain, and the communication between the mobile nodes and the Access Node is proprietary. The full complexity of the MTS is not needed and ACL messages can be transported in an efficient form over the wireless link. Appendix D summarises the optimization of the FIPA-OS agent platform.

<table>
<thead>
<tr>
<th>Distributed FIPA-OS</th>
<th>Clients running FIPA-OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Simplicity in client reduces requirements for performance and memory, emphasises data communication between client and server. A minimal implementation of MTS is needed on the device.</td>
<td>- More memory and computing power is needed, this diminishes wireless use. A full implementation of the MTS is also needed on the device.</td>
</tr>
<tr>
<td>- Since AMS, ACC and DF are located on a single host, this host may become a bottleneck for scalability. This design is tied to an entry-point for CRUMPET services (the Access Node). The scalability can be improved by further distribution of components.</td>
<td>+ This approach is scalable, the full FIPA-OS systems may interact with any other platforms. They may employ an access node, but this is not necessary.</td>
</tr>
<tr>
<td>+ We may use partially dynamic configuration to find the software configuration that best suits the current needs and constraints.</td>
<td>+ We may use partially dynamic configuration to find the software configuration that best suits the current needs and constraints.</td>
</tr>
</tbody>
</table>
5. Conclusions

This document has presented several possible design alternatives that may be used in the development of the small footprint CRUMPET agent platform, the MicroFIPA-OS. In order to have a scalable, interoperable and adaptable software platform for the CRUMPET agents that interact with the tourism services on the network, an agent platform that relies on profile based information is needed. The design of the MicroFIPA-OS consists of several improvements and optimizations to the current FIPA-OS platform that is based on Java 1.2 and intended to be used in a traditional computing environment.

First of all, the FIPA-OS is ported to the PersonalJava programming environment and deployed on a Compaq iPAQ running Linux. The iPAQ reflects the current high-end PDA that supports different kinds of wireless links (GSM, GPRS, WLAN), positioning (GPS) and has enough processing power to support a middleware layer such as Java and MicroFIPA-OS. Linux is a compact and most importantly open operating system.

The current FIPA-OS code is simplified by introducing lightweight components that may be used instead of the original components. Lightweight plug-ins include transport protocol implementations, parsers, task and conversation management, conversations, tasks. These components may be used in future FIPA-OS releases. The available components are defined in agent platform profiles. Hardware profiles define the system performance, and output and input capabilities. Agent platform profiles define the current limitations on the number of tasks, what kind of transport components should be used, what interaction protocols may be used etc.

It is envisaged that the functionality of the MicroFIPA-OS platform may be split between the mobile node and the fixed network. This specification imposes no set boundary, but rather allows the implementation of a scalable architecture that can support partitioning to a varying degree depending on the applications and how much memory and processing power is needed for other tasks. The middleware system should not take too much processing power and memory from the actual applications (which consist of agents).

The distribution of agents was discussed in section 4. With the help of the profile information the system may decide to transfer some functionality to the Access Node. Different kinds of issues affect how the platform should be partitioned between the mobile node and the network: link speed and quality, what kind of services are going to be executed and what are their requirements etc. The clearest choice is that between running the platform agents on the small device or using them on the Access Node.

We have identified several different scenarios for the distribution of functionality between the mobile nodes and the fixed network. These can be condensed into three cases starting from the most simple and ending in the most sophisticated: a minimal system running just the user interface for CRUMPET services (one agent, minimal agent communication), distributed platform that uses the Access Node system agents for communication and maintenance, and full FIPA-OS running on the terminal. Each of these scenarios have their uses, and it is envisaged that the MicroFIPA-OS would cater for all of these scenarios if needed, at least to some degree. However, at first the system would use a less elaborate setup and then proceed into more advanced configurations.
References


http://java.sun.com/docs/white/platform/javaplatformTOC.doc.html


ftp://ftp.omg.org/pub/docs/formal/00-11-03.pdf


http://java.sun.com/products/personaljava/


http://java.sun.com/j2me/


http://www.wapforum.org/
Appendix A: Java Platforms Overview

The current Java platform solutions for small devices can be separated into three categories:

1. **Standard Java based solutions**

The benefits of the Java platform with all the packages: CORBA, RMI, RMI-IIOP, JNI, AWT, SWING, JavaBeans depending on the JDK version. Software can be developed using various development environments. Several manufacturers have their own versions of various JDK code bases for small devices:

- The EPOC implementation (JDK1.1.4)
- IBM J9 VM, VisualAge Micro Edition. Based on JDK 1.2.2. (Linux/StrongARM, PalmOS, WinCE/ARM,MIPS..)

2. **PersonalJava**

A scaled down version of the standard Java (currently 1.1.8 with PJava 1.2 spec) platform. JavaBeans are supported. RMI and security are optional, no CORBA support, JNI and scaled down version of AWT. Sun provides an emulation environment for testing the adherence to the PersonalJava specification. This can be also used to trim the GUI.

- Sun implementation for Windows CE 2.11.
- Sun is planning to release a runtime for StrongARM (the end of October 2000).
- Kaffe (PJava 1.1.1) for various platforms (RMI optional package not implemented)
- Jeode (PJava 1.1.1, PJava 1.2, MIPS, ARM)
- CrEme (MIPS,StrongARM)

3. **J2ME CLDC**

For limited devices. No JNI, RMI, AWT. Native methods are implemented at the virtual machine level (KVM). The VM needs to be recompiled / changed for each device. Communications are abstracted with the Connector-paradigm. The source code is not necessarily compatible with PersonalJava or Standard Java. Motivation: Dynamic content to mobile devices. Current implementations:

- Reference implementation for Palm (proprietary device API).
- Mobile Information Device Profile reference implementation for Win32 and Solaris.

Issues that separate the different Java solutions:

- Virtual Machine footprint and limitations. VM target environments.
- Optional / mandatory packages.
- UI features (SWING/AWT/Subset of AWT/Native).
- Implementation of native methods and interfaces to OS / hardware (for example GPS devices).
- Portability of code.

**Java, Operating Systems and Devices**

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Java solution</th>
<th>RMI</th>
<th>AWT or SWING</th>
<th>JNI</th>
<th>Emulator</th>
<th>Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>PalmOS</td>
<td>J2ME CLDC, proprietary UI, also J9</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>PalmPilot III and V</td>
</tr>
<tr>
<td>EPOC</td>
<td>JDK 1.1.4, 100% pure Java</td>
<td>X</td>
<td>BOTH</td>
<td>X</td>
<td></td>
<td>Ericsson MC 218, Psion series 7, Psion netBook</td>
</tr>
<tr>
<td>WindowsCE</td>
<td>PersonalJava, Kaffe, CrEme, Jeode, J9</td>
<td>X</td>
<td>AWT</td>
<td>X</td>
<td>Sun PJava emulator.</td>
<td>Casio Cassiopeia, Philips Nino</td>
</tr>
<tr>
<td>PocketPC (CE 3.0)</td>
<td>PersonalJava, CrEme, Jeode, (Kaffe)</td>
<td>X</td>
<td>AWT</td>
<td>X</td>
<td>Sun PJava emulator.</td>
<td>Compaq iPAQ H3630 (PJava soon), Casio Cassiopeia E115, HP Jordana 545</td>
</tr>
<tr>
<td>PocketLinux</td>
<td>Kaffe</td>
<td>BOTH</td>
<td>X</td>
<td></td>
<td>Sun PJava emulator.</td>
<td>Compaq iPAQ H3630, Casio Cassiopeia E115</td>
</tr>
<tr>
<td>Windows/ Solaris</td>
<td>J2ME CLDC MIDP</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>Future smart phone (reference implementation)</td>
</tr>
</tbody>
</table>

1 Kaffe does not fully support RMI yet.
**Appendix B: Dynamic Profiles for FIPA-OS**

The aims of the proposed changes to the profile handling classes highlighted in this section are:

- To enable a broader range of options with regard to an Agents configuration to be controlled.
- To allow dynamic extension of the profile types handled by the profiling mechanism.

This will enable:

- An Agents configuration at deployment-time to be better tailored for the device it is executing on (like PDAs). For example for small devices hardware profile can be implemented, so the agent can have more details of the environment it is working in (like screen characteristics).

- Extended user profile types to be simply incorporated into an agent’s profile, so that the agent profile will act as a base profile that can have references to other profiles. This way the profiles can be distributed across the network allowing them to be shared or accessed from multiple remote locations.

In particular, it is envisaged that the AgentProfile class is extended to enable it to encapsulate the other profiles required by an Agent. In addition to this, the use of generic data-binding techniques will be introduced to FIPA-OS, which can be used for a variety of purposes beyond the conversion of XML profiles into objects and vice-versa.

**1 DatabinderObjectProvider Class**

This chapter describes the DatabinderObjectProvider class that is used to locate objects instances in a generic manner.
1.1 CLASS DIAGRAM

DatabinderObjectProvider
(from databinding)

getGlobalObject(type : String) : Object
getObject(type : String) : Object
addDatabinderObjectProvider(dop : DatabinderObjectProvider) : void
removeDatabinderObjectProvider(dop : DatabinderObjectProvider) : void

DefaultProfileObjectProvider
(from profile)

getProfile()

ACCProfile
(from profile)

Profile
(from profile)

AgentProfile
(from profile)

DatabaseProfile
(from profile)

PlatformProfile
(from profile)

RemoteAgentPlatformProfile
(from profile)

TaskManagerProfile
(from profile)

ConversationManagerProfile
(from profile)

Figure 1: DatabinderObjectProvider / DefaultProfileObjectProvider Class Diagram

DatabindingObjectProvider is an abstract class in package fipaos.parser databinding and its purpose is to provide an API for creating unpopulated object instances.

1.2 DESIGN PATTERNS USED

This design is based upon a Factory-like pattern whereby an API is provided behind which a (potentially unknown) implementation of a class which produces instances of objects based upon certain criteria given to it, which itself can be swapped for other implementations. The advantages of this approach are:

- A simple API is provided to developers wishing to load/create Profile instances.
- The particular mechanisms used to obtain an instance are hidden from the developer (instances may need to be created in a particular manner).
- A user-defined mechanism for creating instances can be added to the default mechanism at runtime (the most recently added DatabinderObjectProvider is queried first, then the others).
• User-defined mechanisms and default mechanisms are separated such that changes to one shouldn’t affect the other.

1.3 ABSTRACT METHODS

The following are the methods defined by the DatabinderObjectProvider abstract class for implementation by sub-classes.

public abstract Profile getObject(String)

This method is used to get an instance of a certain object type.

1.4 STATIC (API) METHODS

The following are the methods defined by the DatabinderObjectProvider abstract class that can be used as global API calls.

public static void addDatabinderObjectProvider(DatabinderObjectProvider)

This method adds a DatabinderObjectProvider to the list of available DatabinderObjectProvider within the JVM.

public static void removeDatabinderObjectProvider(DatabinderObjectProvider)

This method removes the given DatabinderObjectProvider.

public static Object getGlobalProfileProviderProfileByName(String)

This method returns a profile from the global ProfileProvider using the profiles name. The name can be either a filename, or an URL type supported by the JRE.

public static Object getGlobalObject(String)

This method returns an object from the registered DatabinderObjectProvider’s using the objects’ type. The type is usually mapped to a Java class name. If these attempts fail, ‘null’ is returned.

1.5 DEFAULTPROFILEOBJECTPROVIDER

DefaultProfileObjectProvider is a concrete implementation of the DatabinderObjectProviderclass bundled with FIPA-OS, which specialises in locating profile objects. Calls to getObject(String type) will result in the following look up steps:

• Central types, like ‘ConversationManager’, ‘MTS’ and ‘ACC’, will be recognised and the appropriate Profile sub-classes automatically instantiated.

• The class name will be guessed based upon the profile name (e.g. if a profile of type “SomeProfileType” is required, the DefaultProfileProvider will attempt to load a class of name “fipaoos.agent.profile.SomeProfileTypeProfile”).

• If neither of the previous steps yields a result, null will be returned.

2 Profile Class

This chapter describes the generic Profile class that is used to represent profiles.
The new Profile class will be an abstract class in package fipaos.agent.profile and its purpose is to act as a base class to all profiles classes.

Unlike the Profile class released as part of FIPA-OS v1.3.2, these profile classes will be simple bean-like containers, which will be populated by a Databinder implementation.

2.2 PROFILE IMPLEMENTATIONS (E.G. CONVERSATIONMANAGERPROFILE)

These classes should simply act as wrappers containing the particular preferences for various parts of an Agent. Below are some suggestions for functionality that could be controlled via various profile types.

- ConversationManager
  - Enable/disable archiving of conversations
- TaskManager
  - Allow tasks to execute in parallel (i.e. mutiple startTask() invocations at once)
  - Enable/disable use of Thread-pooling
- Message Transport Service
  - URL’s for various MTP’s NS’s.
  - Services to use in the MTS’s stack
3 Generic Databinder Architecture

This chapter highlights the generic data-binding architecture to be incorporated into FIPA-OS to enable dynamic profile creation.

Figure 3: Databinding Architecture Overview

Figure 3 highlights the key classes and relationships within a proposed generic data-binding framework. Essentially the DatabindingProvider API has a reference to a “globally referenced” (i.e. via a static reference) DatabinderProvider (by default the DefaultDatabindingProvider).

The global DatabindingProvider provides instances of Databinder objects that are of the required type for a particular Databinding task. DefaultDatabindingProvider is envisaged to be able to create instances of two different types initially:

- XMLDatabinder – provide an XML ↔ Java Object transformation.
- JavaPropertiesDatabinder – provide a “lightweight” data-binder: java properties file ↔ Object transformation

When reconstructing objects, the Databinder implementations should use the DatabinderObjectProvider to instantiate the bean-like classes to encapsulate information. This ensures that differing Databinder implementations use the same concrete classes to represent the same information, rather than using arbitrary/proprietary classes.

The use of the factory-pattern for DatabinderProvider and DatabinderObjectProvider classes ensures that this mechanism is easily extendible by FIPA-OS developers to extend the variety of information that data-binding can be used with, as well as the file-formats available.
Appendix C: Task Mobility

This conceptual design involves distributed task management or partially dynamic partitioning using mobile objects. At agent application start time or agent shell start time on the MicroFIPA-OS the system reads the agent profile and determines what functionality is available and what services can be used on the fixed network. It may not be feasible to implement code mobility, so the system can only make an assessment of the available components and determine which components are feasible to use on the network and which should be instantiated on the device. The device profile may be distributed so that the MicroFIPA-OS loads the FIPA-OS domain configuration from the network-side during startup. This design involves various levels of dynamicity:

1. Run-time assessment of components. What can be instantiated on the device, what may be run on the network. Agent profile determines the functionality that the agents expect to find on the MicroFIPA-OS. Profiles also determine the composition of the MicroFIPA-OS: how many agents and tasks can be executed simultaneously. Agents may need different interaction protocols, tasks (such as DFTask).

2. Partial Dynamic partitioning of tasks. This includes the MicroFIPA-OS fetching new tasks from the network or sending tasks to be performed on the fixed network host at startup (figure 1).

3. Runtime mobile tasks that the MicroFIPA-OS system sends to the host on the fixed network.

DISTRIBUTION OF TASKS

Distributed task and conversation management in figure 1 is done using a stub (or wrapper) on the micro FIPA-OS platform. Tasks are distributed as mobile objects, the task sent to the agent platform that hosts the AMS and DF. This effectively simplifies the micro platform code. When the task is done the object (or resulting object) is returned to the client. This necessitates a TaskServer or similar service on the fixed network side. Conversations, such as fipa-request, can be executed on the terminal. However, more complex conversations should be included into a Task an executed on the fixed network. One problem in this design is that if several agent run on the micro platform and several agents outside the micro platform, should a task communicating with all these agents be executed on the device or on the fixed network.
ConversationServer is not necessarily needed. If we have distributed tasks and tasks support intrinsically conversations.

**Mobile/distributed Tasks; possible designs**

1. Tasks are marshalled or assembled at the micro platform and sent to the fixed network where AMS, DF and ACC reside. TaskServer executes these active objects and sends the return values. Either the whole object is returned (which has an overhead) or only the result is returned.

2. Other way to do this is that when the micro agent starts, it reads the tasks from the profile and registers them to the fixed network side (start-time partitioning). The task objects are sent to the fixed network side for processing. This is done only once per micro agent. Thus the agent has a reserve or supply of pre-registered tasks at the fixed network. These may then be used. The tasks are programmed in such a way that the result is sent to the micro agent, for example in XML or some other suitable format. One interesting issue is XML-RPC, which facilitates remote procedure calls using HTTP and XML. The method calls and return values are defined using XML. The micro platform might use this to distribute functionality.

**UML Diagram: Basic Agent Relationships**

![UML Diagram](image-url)
**RELATED WORK**

This section presents work that has been carried out to use mobile agents on small devices. TACOMA is a platform for mobile agents, and it has been ported to Windows CE and PalmOS (PalmPilot Pro) [JJ097]. This resulted in the design and a C-language based implementation of the TACOMA Lite architecture. Main modifications to the original TACOMA codebase were:

1. Improving the reliability of TCP/IP communication support by adding sleep-periods after socket based errors. For example: implementation of two timers for the send data method. The first timeout specifies the amount of time to wait if a socket error happens. The second timeout is used when we keep getting repeated errors, to give other threads more access to the network and to prevent buffer overflows. The TACOMA Lite team found that these improvements improved the system, and the removal of the timeouts in the send and receive methods added 20% more frequent serious errors. However, determining the correct values for the timers is problematic and depends on several parameters.

2. Improving memory allocation of the PDA by adding robustness to the code – the application should not fail when the memory runs out. Here we can also use sleeping to give other programs the opportunity to free memory. The TACOMA Lite relAlloc method tries to reserve memory, if the method is unsuccessful in doing this it waits for some time and retries. If n retries fail the method returns an error. If the memory allocation fails, the user is prompted to release memory by stopping other programs. On Windows CE the timeout is determined from the number of active threads. PalmOS does the prompting of the user automatically.

3. Minimizing stack usage in the original TACOMA API. This was done by decreasing the number of local function variables and making the methods more self-contained. The resulting API consisted only of half the functions.

A set of example programs were implemented with the TACOMA Lite, one of them being the Communicator, which listen for messages and displays them to the user. Other examples were Weather Alarm and the Stock Ticker. These applications tend to put complexity into the fixed network servers instead of the PDA. The TACOMA Lite architecture relies on a fixed network host, termed Hostel, which acts as a permanent storage and proxy for the PDA.
Appendix D: List of FIPA-OS Optimizations

The following list presents the possible FIPA-OS optimizations. The actual implementation of MicroFIPA-OS may not use all of these implementations, if the current solutions work with the constrained environment.

1. General Optimizations

1.1 Minimize the number of messages on the wireless link, because of long roundtrip times.

1.2 Parsing

1.2.1 Avoid excess parsing, both local and the parsing of incoming messages.

1.2.2 If necessary, use hardcoded-parsing code.

1.2.3 Employ efficient means of communication between the FIPA system components. If the internals are local, use method calling.

2. TaskManager

2.1 Thread pooling

2.2 Limit the number of active tasks (profile based information).

2.3 Re- implement frequently used tasks, such as DFTask to be more efficient.

3. ConversationManager

3.1 Limit the number of active conversations (same as 2.2).

3.2 Disable conversation logging

3.3 Implement limited versions of ConversationManager

3.3.1 The available ConversationManagers are defined in the platform profile.

3.3.2 Minimal ConversationManager, fipa-request, one-shot protocols

3.4 MTS

3.4.1 Simplified message stacks.

3.4.2 Message buffering.

3.4.3 At the extreme, hardcode the MTS functionality at the mobile host.

4. Architectural Issues

4.1 Profiles

4.2 Support for distributed profiles (profile management)

4.3 Support for external and local service configuration information
4.4 Full Platform

4.4.1 Access Node discovery using static address in platform profile.

4.4.2 Registration and deregistration from the Access Node.

4.4.3 Message buffering at the Access Node (ACC).

4.5 Partitioned Platform

4.5.1 Transfer ACL and parsing to fixed network and use compact object representations as the internal format in the platform.

4.5.2 MTS at the Access Node

4.5.2.1 FIPA MTS support for mobile “partial” platforms.

4.5.2.2 Support for agent naming in the CRUMPET domain and message routing to mobile hosts.

4.5.2.3 Message buffering at MTS.

4.5.3 Mobile host

4.5.3.1 Simplified implementation of MTS.

4.5.3.2 Support for disconnected operation.

4.5.4 General issues

4.5.4.1 Proprietary interface between mobile and fixed network nodes.

4.5.4.2 ACC, AMS and DF at the fixed network.

4.5.4.3 DFs may also be available on the mobile nodes.
Appendix E: iPAQ Hardware Specification

**Overview:** The iPAQ H3630 from Compaq is a small and powerful handheld computer based on the StrongARM processor running at 206 MHz. It provides large memory, a high quality color screen (320x240) and a long-life Lithium-ion battery in a very compact handheld package. It also features a clip-on expansion-pack architecture to provide hardware expandability. Current expansion-pack architecture does not support the use of PCMCIA and CompactFlash cards at the same time.

**Specification:** Compaq iPAQ H3600 Hardware Design Specification - Version 0.2f
http://www.handhelds.org/Compaq/iPAQH3600/iPAQ_H3600.html

<table>
<thead>
<tr>
<th>Specification</th>
<th>IPAQ H3630</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platform / OS</strong></td>
<td>Windows Powered Pocket PC or Linux (in Flash memory)</td>
</tr>
<tr>
<td><strong>Processor</strong></td>
<td>206 MHz Intel StrongARM SA-1110 32-bit RISC Processor</td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td>32 MB</td>
</tr>
<tr>
<td><strong>ROM</strong></td>
<td>16 MB FLASH ROM</td>
</tr>
<tr>
<td><strong>Display</strong></td>
<td>24 mm Pixel Pitch, 4096 color, Color reflective thin film transistor (TFT) LCD</td>
</tr>
<tr>
<td><strong>Resolution</strong></td>
<td>240 x 320, Image Size: 2.26 x 3.02 inches</td>
</tr>
<tr>
<td><strong>Battery/Working hour</strong></td>
<td>950 mAh Lithium Polymer (up to 12hrs of life)</td>
</tr>
<tr>
<td><strong>Infrared port</strong></td>
<td>IrDa standard, 115KB/s</td>
</tr>
<tr>
<td><strong>USB support</strong></td>
<td>Yes (USB cable/cradle needed)</td>
</tr>
<tr>
<td><strong>Audio-in jack</strong></td>
<td>Yes – Stereo</td>
</tr>
<tr>
<td><strong>Speaker &amp; Voice Recorder</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Card slot</strong></td>
<td>Optional expansion packs: CompactFlash Type I and II, and PC Card (PCMCIA)</td>
</tr>
<tr>
<td><strong>Dimensions</strong></td>
<td>130 x 15.9 x 83.5 mm</td>
</tr>
<tr>
<td><strong>Weight</strong></td>
<td>170 g</td>
</tr>
<tr>
<td><strong>Operating humidity</strong></td>
<td>20% to 80%</td>
</tr>
<tr>
<td><strong>Operating temperature</strong></td>
<td>0° to 40°C</td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
<td>Wireless LAN PC card, CompactFlash modem card, CompactFlash memory cards, other PCMCIA and CompactFlash cards.</td>
</tr>
</tbody>
</table>