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

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Contractual Date of Delivery to CEC:</td>
<td>31/11/2001</td>
</tr>
<tr>
<td>Actual Date of Delivery to CEC:</td>
<td>21/12/2001</td>
</tr>
<tr>
<td>Title of Deliverable:</td>
<td>Network Management and Control Agents Implementation</td>
</tr>
<tr>
<td>WP Contributing to Deliverable:</td>
<td>WP3</td>
</tr>
<tr>
<td>Nature of Deliverable:</td>
<td>Report</td>
</tr>
<tr>
<td>Authors:</td>
<td>Mikko Laukkanen (Sonera), Heikki Helin (Sonera), Heimo Laamanen (Sonera), Barbara Schmidt-Belz (FIT).</td>
</tr>
</tbody>
</table>

**Abstract:** In this Deliverable we describe the nomadic application support implementation, which is an agent-based middleware providing means for building adaptive applications for nomadic users in CRUMPET. The overall architecture of the nomadic application support in CRUMPET consists of two main entities: the mobile device, which in CRUMPET is Compaq iPAQ running Linux operating system, and the CRUMPET Access Node. CRUMPET architecture follows the client-mediator-server paradigm where mobile devices are the clients, which are wirelessly connected to the services residing on the fixed network via a CRUMPET Access Node. CRUMPET Access Node is an entity in the fixed network, which hosts a FIPA-OS agent platform being able to mediate agent interactions between the wireless and wired worlds. The implementation consists of two agents on both mobile device and Access Node for monitoring and controlling the wireless link, ontology, and an efficient way for agent communication over the wireless link. The ontology defines a QoS terminology and methods to access the services of monitor and control agents. The efficient way for agent communication consists of a bit-efficient encoding of agent messages and a message transport protocol designed for wireless links.

**Keyword List:** multi agents, software agents, FIPA, FIPA-OS, nomadic application

Copyright © 2000 - 2001 by Partners of CRUMPET
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 specifies the Nomadic Application Support (NAS) implementation in CRUMPET, and especially concentrates on the usage of the NAS implementation and on explaining the differences between the implementation and the design in the Deliverable 3.1.

The overall architecture of the NAS in CRUMPET consists of two main entities: the mobile device (which in CRUMPET will be Compaq iPAQ running Linux operating system) and the CRUMPET Access Node. These two entities are connected through a wireless link. CRUMPET architecture follows the client-mediator-server paradigm where mobile devices are the clients, which are wirelessly connected to the services residing on the fixed network via a CRUMPET Access Node. CRUMPET Access Node is an entity in the fixed network, which hosts a FIPA-OS agent platform being able to mediate agent interactions between the wireless and wired worlds.

The implementation described in this Deliverable is based on FIPA Nomadic Application Support specification. The implementation consists of two agents: Monitor Agent and Control Agent. Monitor Agent monitors the Quality of Service (QoS) on the wireless link and is able to both respond to queries about QoS and take in subscriptions for changes in QoS so that when QoS changes the subscribed agents are informed about it.

The Control Agent controls the use of the wireless link. It is able both to autonomously establish and shutdown connections based on the QoS on the wireless link, or to establish and shutdown connections on request. The Control Agent provides an ACL interface for other agents to access its services. It also maintains profiles for network access. The Control Agent in CRUMPET supports GSM, HSCSD, WLAN, and GPRS. UMTS will be supported only if there will be a running network and terminals available well before the CRUMPET trials.

Deliverable 3.1 specified that the protocol used on the wireless link is WAP as specified by FIPA. Due to the lack of available WAP stack implementations we decided not to support WAP. Instead, we designed and implemented our own message transport protocol to be used on the wireless link. The protocol is written in pure Java, is reliable and is able to recover from disconnections. The ACL messages and message envelopes are encoded bit-efficiently according to the FIPA specifications. In the case of disconnection on the wireless link, the messages are buffered so that when the link is re-established, the buffered messages are transmitted and the session continues.
## Version history

<table>
<thead>
<tr>
<th>Revision</th>
<th>Modified by</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Mikko Laukkanen</td>
<td>08.10.01</td>
<td>Initial version</td>
</tr>
<tr>
<td>0.2</td>
<td>Mikko Laukkanen</td>
<td>29.11.01</td>
<td>Almost completely rewritten.</td>
</tr>
<tr>
<td>0.3</td>
<td>Mikko Laukkanen</td>
<td>30.11.01</td>
<td>Minor modifications.</td>
</tr>
<tr>
<td>0.4</td>
<td>Mikko Laukkanen</td>
<td>07.12.01</td>
<td>Corrected according to comments by Heikki and Heimo. Added some comments.</td>
</tr>
<tr>
<td>0.5</td>
<td>Mikko Laukkanen</td>
<td>07.12.01</td>
<td>Initial version for CRUMPET partners.</td>
</tr>
<tr>
<td>1.0</td>
<td>Mikko Laukkanen</td>
<td>21.12.01</td>
<td>Corrected according to comments from Barbara and Heimo. Ready to be delivered to EU.</td>
</tr>
</tbody>
</table>
### ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Agent Communication Channel</td>
</tr>
<tr>
<td>ACL</td>
<td>Agent Communication Language</td>
</tr>
<tr>
<td>AMS</td>
<td>Agent Management System</td>
</tr>
<tr>
<td>AWT</td>
<td>Abstract Windowing Toolkit</td>
</tr>
<tr>
<td>BE</td>
<td>Bit-efficient</td>
</tr>
<tr>
<td>CA</td>
<td>Control Agent</td>
</tr>
<tr>
<td>CAN</td>
<td>CRUMPET Access Node</td>
</tr>
<tr>
<td>CM</td>
<td>Congestion Manager</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>DF</td>
<td>Directory Facilitator</td>
</tr>
<tr>
<td>FIPA</td>
<td>Foundation for Intelligent Physical Agents</td>
</tr>
<tr>
<td>FIPA-OS</td>
<td>FIPA-Open Source</td>
</tr>
<tr>
<td>GPRS</td>
<td>General Packet Radio Service</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile Communications</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HSCSD</td>
<td>High Speed Circuit Switched Data</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IOP</td>
<td>Interoperable Inter-ORB Protocol</td>
</tr>
<tr>
<td>IMTS</td>
<td>Intra-platform Message Transport Service</td>
</tr>
<tr>
<td>ISP</td>
<td>Internet Service Provider</td>
</tr>
<tr>
<td>J2ME</td>
<td>Java 2 Micro Edition</td>
</tr>
<tr>
<td>J2SE</td>
<td>Java 2 Standard Edition</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Kit</td>
</tr>
<tr>
<td>JINI</td>
<td>Java Intelligent Network Infrastructure</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>MA</td>
<td>Monitor Agent</td>
</tr>
<tr>
<td>MBS</td>
<td>Message Buffering Service</td>
</tr>
<tr>
<td>MD</td>
<td>Mobile Device</td>
</tr>
<tr>
<td>MTC</td>
<td>Message Transport Connection</td>
</tr>
<tr>
<td>MTP</td>
<td>Message Transport Protocol</td>
</tr>
<tr>
<td>MTS</td>
<td>Message Transport Service</td>
</tr>
<tr>
<td>NAS</td>
<td>Nomadic Application Support</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Transformation</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>PCMCIA</td>
<td>PC Memory Card International Association</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>PDU</td>
<td>Protocol Data Unit</td>
</tr>
<tr>
<td>PJava</td>
<td>Personal Java</td>
</tr>
<tr>
<td>PP</td>
<td>Presentation Performer</td>
</tr>
<tr>
<td>PPP</td>
<td>Point-to-Point-Protocol</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>SL</td>
<td>Semantic Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunications Systems</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WAE</td>
<td>Wireless Application Environment</td>
</tr>
<tr>
<td>WAP</td>
<td>Wireless Access Protocol</td>
</tr>
<tr>
<td>WDP</td>
<td>Wireless Datagram Protocol</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless Local Area Network</td>
</tr>
<tr>
<td>WMTP</td>
<td>Wireless Message Transport Protocol</td>
</tr>
<tr>
<td>WSP</td>
<td>Wireless Session Protocol</td>
</tr>
<tr>
<td>WTP</td>
<td>Wireless Transaction Protocol</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Mark-up Language</td>
</tr>
</tbody>
</table>
### GLOSSARY

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Access Point</strong></td>
<td>Provides wireless access to the CRUMPET Access Node.</td>
</tr>
<tr>
<td><strong>Agent shell</strong></td>
<td>The basic building block of agents in FIPA-OS that provides services such as message passing, tasks and conversations.</td>
</tr>
<tr>
<td><strong>AppAgent</strong></td>
<td>Any CRUMPET agent other than MA or CA.</td>
</tr>
<tr>
<td><strong>Bluetooth</strong></td>
<td>A short-range (up to 10 metres) 2.4GHz wireless connectivity solution allowing a creation of wireless Personal Area Networks (PANs) for data exchange between devices such as notebook and pocket computers.</td>
</tr>
<tr>
<td><strong>CA</strong></td>
<td>Controls the MTC and MTP properties according to the FIPA Nomadic Application Support specification.</td>
</tr>
<tr>
<td><strong>Content Adaption Service Agent (CASA)</strong></td>
<td>Takes care of adapting the data coming from the fixed network onto the wireless link according to the current QoS of the wireless link.</td>
</tr>
<tr>
<td><strong>CRUMPET Client Agent</strong></td>
<td>CRUMPET User Agent taking care of user interface to the CRUMPET services.</td>
</tr>
<tr>
<td><strong>CRUMPET Access Node</strong></td>
<td>The CRUMPET Access Node is a mediator between the mobile device and the CRUMPET service agents located in the fixed network.</td>
</tr>
<tr>
<td><strong>CRUMPET Access Node FIPA-OS</strong></td>
<td>Full-blown FIPA-OS supporting also WAP MTP.</td>
</tr>
<tr>
<td><strong>FIPA</strong></td>
<td>Foundation for Intelligent Physical Agents (FIPA) is an international organisation that aims to standardise interoperable software agent architecture.</td>
</tr>
<tr>
<td><strong>FIPA-OS</strong></td>
<td>FIPA-OS is an open-source software agent platform that complies with the FIPA specifications.</td>
</tr>
<tr>
<td><strong>IMTS</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>iPAQ</strong></td>
<td>Compaq iPAQ is a family of personal devices. In this Deliverable with iPAQ we refer to iPAQ H3600, which is a hand-held computer.</td>
</tr>
<tr>
<td><strong>IrDA</strong></td>
<td>Infrared Data Association. The industry organisation of computer, component, and telecommunications vendors who have established the standards for infrared communication between computers and peripheral devices such as printers.</td>
</tr>
<tr>
<td><strong>J2ME</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>J2SE</strong></td>
<td>Java 2 Standard Edition is the traditional Java application environment.</td>
</tr>
<tr>
<td><strong>Kaffe</strong></td>
<td>Kaffe is a clean room implementation of the PersonalJava specification.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Linux</strong></td>
<td>Linux is a Unix-based open source operating system.</td>
</tr>
<tr>
<td><strong>MA</strong></td>
<td>Monitor Agent. Monitors the link properties according to the FIPA Nomadic Application Support specification.</td>
</tr>
<tr>
<td><strong>MicroFIPA-OS</strong></td>
<td>The CRUMPET agent execution environment for small devices that is based on the FIPA-OS agent platform.</td>
</tr>
<tr>
<td><strong>Micro-MTS</strong></td>
<td>Lightweight FIPA-OS MTS designed for small-footprint devices.</td>
</tr>
<tr>
<td><strong>Mobile Device</strong></td>
<td>Mobile device is either a PDA or a laptop computer capable of wireless communication and is used by the end-user for accessing the CRUMPET services.</td>
</tr>
<tr>
<td><strong>Non-agent apps</strong></td>
<td>Any application, which does not use Micro-FIPA-OS nor any other CRUMPET services, but accesses network. Example: Web browser.</td>
</tr>
<tr>
<td><strong>PJava</strong></td>
<td>PersonalJava is a scaled down version of the standard Java 1.1 Virtual Machine (VM) and contains only a subset of the JDK 1.1 APIs. The subset includes Abstract Windowing Toolkit (AWT) with minor changes and classes that provide support for applets and applications.</td>
</tr>
<tr>
<td><strong>RMI</strong></td>
<td>Remote Method Invocation is the Java way of executing distributed remote procedure calls.</td>
</tr>
<tr>
<td><strong>WAP Gateway</strong></td>
<td>WAP gateway is an entity translating requests from the WAP protocol stack to the WWW protocol stack and translating WAP content into compact encoded formats to reduce the size of data over the network.</td>
</tr>
</tbody>
</table>
## Table of contents

1 **INTRODUCTION** ................................................................................................................................. 9

2 **ARCHITECTURE FOR SUPPORTING NOMADICITY** ............................................................................. 10

3 **NAS ONTOLOGY** .................................................................................................................................. 12

4 **MONITOR AGENT** .................................................................................................................................. 14

4.1 **GETTING CURRENT QoS** ...................................................................................................................... 15

4.2 **SUBSCRIBING TO CHANGES IN QoS** ................................................................................................... 16

4.3 **QUALITY OF INFORMATION GATHERING USING NETTIMER** .............................................................. 17

4.3.1 **Using Nettimer instead of Congestion Manager** .................................................................................. 17

4.3.2 **How Nettimer is used by Monitor Agent** ............................................................................................. 18

4.3.3 **How the Nettimer is used in Mobile Device and the Access Node** ......................................................... 18

5 **CONTROL AGENT** .................................................................................................................................. 20

5.1 **REQUEST-BASED OPERATION** ............................................................................................................... 20

5.2 **REQUESTING CA TO OPEN COMMUNICATION CHANNEL** ....................................................................... 21

5.3 **REQUESTING CA TO CLOSE COMMUNICATION CHANNEL** ...................................................................... 22

5.4 **REQUESTING CA TO ACTIVATE MESSAGE TRANSPORT PROTOCOL(S)** .................................................. 23

5.5 **REQUESTING CA TO DEACTIVATE MESSAGE TRANSPORT PROTOCOL(S)** .............................................. 24

5.6 **AUTONOMOUS OPERATION** .................................................................................................................. 25

5.7 **SUPPORT FOR ROAMING** ....................................................................................................................... 25

6 **EFFICIENT AGENT COMMUNICATION** .............................................................................................. 27

6.1 **COMMUNICATION ARCHITECTURE IN CRUMPET** ................................................................................. 27

6.2 **BIT-EFFICIENT ACL MESSAGE** ........................................................................................................... 28

6.3 **BIT-EFFICIENT ENVELOPE** ................................................................................................................ 28

6.4 **MESSAGECODEC** .................................................................................................................................. 28

6.5 **WIRELESS MESSAGE TRANSPORT** ...................................................................................................... 28

7 **DISCONNECTION HANDLING AND BUFFERING** ................................................................................. 33

8 **CONFIGURATION FILES** ....................................................................................................................... 35

8.1 **NAS CONFIGURATION PROFILE** .......................................................................................................... 35

8.2 **NETWORK CARD PROFILE** ................................................................................................................. 35

8.3 **ACCESS POINT PROFILE** ................................................................................................................... 36

8.4 **LOGIN INFO PROFILE** ......................................................................................................................... 37

9 **EXAMPLE USAGE OF NOMADIC APPLICATION SUPPORT** .............................................................. 38

9.1 **LOGIN PROCEDURE** ................................................................................................................................ 38

9.2 **ACCESSING ADAPTED SERVICES** ......................................................................................................... 39

10 **NAS DEPLOYMENT** ............................................................................................................................. 40

10.1 **PREREQUISITES** .................................................................................................................................. 40

10.2 **DIRECTORY STRUCTURE** .................................................................................................................... 40

10.3 **PACKAGE NAS.BE** .............................................................................................................................. 41

10.4 **PACKAGE NAS.CA** ............................................................................................................................ 41

10.5 **PACKAGE NAS.MA** ............................................................................................................................ 41

10.6 **PACKAGE NAS.ONTOLOGY** ................................................................................................................ 42

10.7 **PACKAGE NAS.PARSER** ..................................................................................................................... 42

10.8 **PACKAGE NAS.TRACE** ....................................................................................................................... 42

10.9 **PACKAGES FIPAOS AND MICROFIPAOS** ........................................................................................... 42

10.10 **BUILDING AND INSTALLING THE DISTRIBUTION** ............................................................................ 43

10.11 **CONFIGURING THE DISTRIBUTION** .................................................................................................. 43

10.12 **RUNNING THE DISTRIBUTION** .......................................................................................................... 43

10.13 **MAINTENANCE** ................................................................................................................................... 43
1 INTRODUCTION

The overall aim of CRUMPET [CRU01a] 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 is based on a standards-compliant open source agent framework, extended to support nomadic applications, devices, and networks. The implementation is developed using best practice, including “open source for open standards” development, developing generic re-usable components on an open distributed architecture. CRUMPET software agent environment is based on an open-source FIPA compliant agent platform called FIPA-OS [Pos99] and its variation, MicroFIPA-OS [CRU00b, CRU01b], for handheld devices.

Wireless wide-area networks are in a phase of rapid development. High Speed Circuit Switched Data (HSCSD) and General Packet Radio Service (GPRS) are already in the market and the Universal Mobile Telecommunications System (UMTS) is expected to be launched in 2003-4. Utilizing wireless LANs (WLAN) as public ‘hot spot’ access networks to the Internet is currently under investigation and implementation.

Each of these networks has its specific data transmission characteristics. However, the basic challenge of wireless wide-area communications is as follows: In the environment of wireless data communications the Quality of service (QoS) (such as line rate, delay, throughput, round-trip time, and error rate) may change dramatically when a user moves from one location to another. For example, when the user roams from a UMTS cell to a GPRS cell, the throughput may drop from 1 Mbits/s down to 24 Kbits/s. In addition, it is foreseen that seamless roaming between different network technologies (e.g. between UMTS and WLAN) will be needed in the near future, leading to an increase in the variability mentioned above. This high variety, high volatility environment of Internet-based services creates a need for adaptability. Users will demand services that will automatically and transparently adapt to the changes mentioned above.

In this Deliverable we describe nomadic application support implementation, which provides means for building adaptive applications for nomadic users. The nomadic application support (NAS) implementation is divided into four major parts: monitoring and controlling a wireless link, a NAS ontology and efficient agent communication. Monitoring and controlling of the link is carried out by a Monitor Agent (MA) and a Control Agent (CA). The NAS ontology defines a vocabulary to access the services of MA and CA. The efficient agent communication consists of bit-efficient encoding for message envelopes and ACL messages as well as a message transport protocol (MTP) designed for a wireless environment. Once NAS-enabled, the agent platform and the agents running on it are provided with tools for adapting to constantly changing QoS, are able to roam also between different networks, and are able to communicate efficiently over the wireless link.
2 ARCHITECTURE FOR SUPPORTING NOMADICITY

The architecture we are working with is shown in Figure 1. Mobile devices access the systems through a wireless network, which can be, for instance, GSM. The access node connects the mobile devices to the services, which can reside anywhere in the fixed network.

![Figure 1: Nomadic architecture in CRUMPET](image)

Mobile devices run a MicroFIPA-OS agent platform, whereas on the access node we can have either MicroFIPA-OS or a full-blown FIPA-OS agent platform. Both MicroFIPA-OS and the platform on the Access Node host MA and CA. In addition, both of the platforms may host additional agents that can use the services of MA and CA. For example, in the CRUMPET architecture we have in the mobile device a client agent to take care of user interactions.

In order to allow the MicroFIPA-OS and the Access Node’s platform to take the wireless environment into account, we have added a number of components to them (see Figure 1). For minimizing the bits transferred over the wireless, message envelopes and ACL messages are encoded bit-efficiently according to FIPA specifications [FIPA00b, FIPA00c]. To be able to recover from sudden disconnections, messages are buffered in a similar way, which is specified in [FIPA01b]. At the lowest level we have a message transport protocol, which is designed for wireless environments.

Our architecture allows a centralization of the handling of the wireless link characteristics into one place, the Access Node, so that the agents in the fixed network do not have to take the characteristics of the wireless communication into account. Thus, agents in the fixed network can use the message transport protocols and connections as usual.

In order to achieve wide range compatibility within FIPA implementations, the design is divided into two parts: the core and the platform-specific part. The core part is common to every FIPA platform, while the platform-specific part has to be configured for each FIPA platform. The configuration specifies how the NAS components (see Figure 1, NAS components are in darker color) are plugged.
into the platform. This basically means that the developer has to implement certain interfaces in order to allow the core part to interact with the platform.

The implementation is mostly Java 1.1-compliant code, because most of the Java Virtual Machines (JVM) do not support versions above this. After studying the available JVMs suitable for our target device, we chose to use Sun JDK1.1.8. There are alternatives though, for instance J2ME CLCD/CDC, a clean-room JVM implementations (Kaffe), and Java 2. Of these, we found that J2ME CLCD is too limited for our purposes, because it does not provide a JNI interface. CDC does not, although having a much smaller footprint, provide sufficient performance enhancements. Kaffe has problems in threading, which may cause deadlocks. Java 2 has too large footprint for handheld devices. The core part is partly written in C, but the platform-specific part is written in pure Java. Furthermore, we have implemented some functionality using third-party software. However, we are not restricted to use these tools, but instead, they can be easily replaced.

On both mobile device and the access node we run Linux. The reason for choosing Linux is that it has a good networking support, it is well configurable, and because it is open-source, we can make modifications to the kernel if needed. Moreover, Linux is available for our target mobile device (Compaq iPAQ H3630). However, we are not ruling out other operating systems, such as PocketPC. Because most of the code is written in Java, porting from one operating system to another is not hard, although not trivial.
3 NAS ONTOLOGY

Our NAS ontology is based on the FIPA Nomadic Application Support specifications [FIPA00a] but we have made modifications and enhancements so that our NAS better serves its users. The modifications and enhancements are as follows:

- Although we support all the QoS parameters, we are not utilising all of them.
- We have added one boolean QoS-parameter called `available`, which indicates, if a wireless network is available or not, because `status` parameter is not expressive enough to indicate that. For instance, `available` can be true even if `status` is `disconnected`.
- We do not support function “use” for selecting an MTP, because we believe that it is not feasible to support multiple MTPs on a handheld device.

We have modeled the ontology as Java objects so that all the complexity is hidden behind a `NasOntologyFactory`, which provides a method for passing in the ACL object. `NasOntologyFactory` is able to create an ontology object out of the content-field of the ACL message. Figure 2 illustrates how this is implemented. We have three logical layers: tokenizers, content parsers and ontology objects. The tokenizer layer contains parsers for different kinds of ontology-field encodings. The tokenize parser performs two functions: it identifies the type of the content field and it outputs the parsed tokens to the content parser layer. Content parsers then create the correct ontology object. For instance, if `XMLContentParser` finds out that the content represents an `OpenCommChannel` function, it creates an `OpenCommChannelParser`, which in turn creates an actual `OpenCommChannelOntology` object out of the tokens.

![Figure 2: NAS ontology model](image)

In the following we show an example fragment of code that demonstrates the use of `NasOntologyFactory`. 
public NasOntology testNasOntologies (String content)
{
    NasOntology returnOnt = null;
    try
    {
        NasOntology ont = NasOntologyFactory.create
        (NasOntologyConstants.XML, content);
        byte[] bytes = ont.toBinaryXML ();
        NasOntology ont2 = NasOntologyFactory.create
        (NasOntologyConstants.BINARY_XML, bytes);
    }
    catch (Exception e)
    {
        System.err.println ("Exception: " + e);
        e.printStackTrace ();
    }
}

This code fragment implements a test method, which takes one String parameter representing a content field of an ACL message. At line 6 and 7 the NasOntologyFactory is used to create a NasOntologyObject out of the String. NasOntologyFactory provides a create method, which needs two parameters: the type of the encoding and the actual content. All the supported encoding types are listed in NasOntologyConstants.

Every NasOntology can be asked to give its content in a specific format. The NAS implementation supports three formats: XML, Binary XML and Java serialized objects. At line 8 the created NasOntology object is asked to give Binary XML representation out of itself. At line 9 and 10 this newly encoded content is again given to NasOntologyFactory for parsing and creating a new NasOntologyObject.

NasOntologyFactory is not able to reason about the format of the content object the create method is given. Although this could be done, we do not support it, because it makes the implementation too complex. Instead we like to keep the implementation simple and we propose that either the entity using NasOntologyFactory reasons about the content format, or the format is decided to be the same either always, on session basis, or on conversation basis.
4 MONITOR AGENT

MA can serve its clients in two different ways. Firstly, it can answer a single query about the current QoS of a wireless link. Secondly, it can inform its client on a subscription basis about changes in the QoS of the wireless link. MA is accessed externally through an ACL interface using NAS ontology as described in the previous section. Internally the platform-specific part accesses the core part of MA using MAFunctions-interface. Having first parsed an incoming ACL message, MA passes the created Ontology object to MAFunctions-interface, which in turn, based on the object, creates and executes a corresponding Function object. This procedure is illustrated in Figure 3.

![Figure 3: Monitor Agent functionality](image)

In the case of a query for the current QoS, MA delegates the acquisition of the QoS to QosManager, which in turn uses Collector-interface to get the current value. Collector-interface is implemented by concrete Collector objects, each specialized in a certain QoS parameter(s). For the actual measurement of the QoS on the wireless link we use a tool called Nettimer [Lai01], which either passively listens to network traffic or actively probes the network. However, implementations of NAS architecture are not restricted to use only these collectors, but any kind of a collector, which suits better for the application area in question can be implemented. This can be done by writing a concrete Collector object that implements the Collector-interface.

In subscription mode an agent may define constraints for the variation in QoS so that when the QoS values violate the constraint, the agent is informed. We use a concept of watermarks to define constraints in the QoS of wireless data communications. The basic idea of the watermarks is described in [Pra01]. It basically allows definition of two kinds of watermarks, high and low, which specify the constraints as an interval between parameter values. Figure 4 illustrates a configuration of watermarks. Each level between watermarks is either a low watermark or a high watermark. A high watermark is violated when a parameter value crosses it from below, and similarly a low watermark is violated when a parameter value crosses it from above. An example of high watermark violation is the parameter value changes from V1 to V2, where the high watermark X3 is violated (see Figure 4). Similarly, when the parameter value changes from V4 to V5, the low watermark X4 is violated, but change from V6 to V5 does not cause any watermark violations. Further, not every watermark violation is treated as an event that triggers a notification to the agent. For example, when the parameter value changes from Vn to Vm crossing a high watermark Wh, the system informs the agent about the current value of the parameter value. Now, if the value changes back to below Wh and then again above Wh, the agent is not informed, because the watermark Wh was not cleared in between. Following [Pra01], a high watermark is cleared if any low watermark below it is violated. Equally, a low watermark is cleared when any high watermark above it is violated. The agent is notified only if a clear watermark is violated. Thus, watermarks provide ways for avoiding jittering effects.
4.1 Getting current QoS

Example of current-qos when requesting current QoS for a given MTC:

Request:

```java
[1] public void sendCurrentQos ()
[2] {
[3]   ACL acl = new ACL ();
[4]   AgentID target = new AgentID ();
[5]   target.setName ("ma-an@localap");
[8]   acl.setReceiverAID (target);
[9]   acl.setPerformative ("query-ref");
[10]  CommChannel commChannel = new CommChannel ();
[11]  commChannel.setName (CAConstants.GSM);
[13]  commChannel.addOption ("some", "option");
[14]  CurrentQosOntology ont = new CurrentQosOntology ();
[15]  ont.setCommChannel (commChannel);
[16]  ont.setAID (myAid);
[17]  ont<Expression ("action");
[18]  acl.setContentObject (ont.toXML ());
[19]  acl.setSenderAID (myAid);
[20]  forward (acl);
[21] }
```
4.2  Subscribing to changes in QoS

Example of QoS subscription:

Request:

```java
public void sendSubscription ()
{
    ACL acl = new ACL ();
    AgentID target = new AgentID ();
    target.setName ("ma@localap");
    target.setAddresses ("(sequence phttp://131.177.251.97/acc)");
    acl.setLanguage (FIPACONSTANTS.CL_REP_XML);
    acl.setReceiverAID (target);
    acl.setPerformative (FIPACONSTANTS.SUBSCRIBE);
    CommChannel commChannel = new CommChannel ();
    commChannel.setName (CACONSTANTS.GSM);
    commChannel.setTargetAddr ("phttp://foo.bar:8080/acc");
    commChannel.addOption ("some", "option");
    Vector qosParameters = new Vector ();
    QosParameter param1 = new QosParameter ();
    param1.setName (MACONSTANTS.THROUGHPUT);
    param1.setUnit (MACONSTANTS.BIT_PER_S);
    QosParameter param2 = new QosParameter ();
    param2.setName (MACONSTANTS.RTT);
    param2.setUnit (MACONSTANTS.MS);
    qosParameters.addElement (param1);
    qosParameters.addElement (param2);
    Vector constraintQosParameters = new Vector ();
    QosParameter constraintParam1 = new QosParameter ();
    constraintParam1.setName (MACONSTANTS.THROUGHPUT);
    constraintQosParameters.addElement (constraintParam1);
    Vector watermarks = new Vector ();
    Watermark wm1 = new Watermark (Watermark.LOW, "rate-value", "bit/s", "8000");
    Watermark wm2 = new Watermark (Watermark.HIGH, "rate-value", "bit/s", "8500");
    watermarks.addElement (wm1);
    watermarks.addElement (wm2);
    SubscribeNotificationOntology ont = new SubscribeNotificationOntology ();
    ont.setCommChannel (commChannel);
    ont.setQosParameters (qosParameters);
    ont.setConstraintQosParameters (constraintQosParameters);
    ont.setWatermarks (watermarks);
    ont.setAID (myAid);
    ont.setExpression ("action");
    acl.setContentObject (ont.toXML ());
    acl.setSenderAID (myAid);
    forward (acl);
}
```
4.3 Quality of information gathering using Nettimer

4.3.1 Using Nettimer instead of Congestion Manager

In Deliverable 3.1 we suggested that Congestion Manager [And00, Bal99] should be used as the primary tool for QoS information gathering. However, Deliverable 3.1 did not rule out other tools, which could either replace or complement Congestion Manager. Congestion Manager is a Linux kernel module, which replaces some functionality in the Linux TCP implementation. Congestion Manager is only available on kernel version 2.2.9 and Compaq iPAQ only supports 2.4.x kernel versions.
Since the release of Deliverable 3.1 a tool called Nettimer has been implemented by Kevin Lai from Stanford University [Lai01]. Nettimer is a program for measuring network link bandwidth. It is a tool for measuring network bandwidth in the same way as ping is a tool for measuring network round trip time.

Nettimer is flexible enough for varying several parameters about what to measure:

1. It can measure all link bandwidths along a path or just the bottleneck link bandwidth,
2. it can measure passively or actively,
3. it can measure one or more hosts, and
4. it takes measurements from traces or measure in real time.

When measuring just the bottleneck link bandwidth along a path, nettimer can either passively listen to other applications’ traffic for measurement, or generate its own probe traffic. Passive measurement does not put additional load on the network. Active measurement is useful when there is no other traffic to measure. When measuring all the link bandwidths along a path, nettimer must use active probing.

In addition, nettimer can take measurements at one or more hosts. Measuring at more hosts allows accurate results at the cost of more effort to deploy software.

Nettimer implements functionality similar to Congestion Manager: it provides real time QoS information. However, Nettimer has a major advantage over Congestion Manager: it operates on Linux user level, thus not needing any modifications to the Linux kernel. This makes porting to other Linux distributions, other end-user devices and platforms a lot easier than it is with Congestion Manager.

Whereas Congestion Manager also provides tools for service differentiation, Nettimer only measures bandwidth. However, we came to a conclusion that service differentiation in CRUMPET is not needed in the Mobile Device and only possibly needed at the Access Node, where we can use other tools, for instance Linux kernel packet filtering techniques. Therefore, we made a decision to use Nettimer instead of Congestion Manager.

4.3.2 How Nettimer is used by Monitor Agent

Nettimer runs in its own standalone process. Nettimer is only able to print the bandwidth information to the terminal, and provides no API for acquiring the bandwidth information. For this reason, we modified the Nettimer so that it dumps all the measured data into UDP datagrams and sends them to the Monitor Agent, which filters, analyses the information, and provides it to other agents. By removing all the libraries (ncurses) that were required for printing the information to the terminal, we could reduce the binary size dramatically; this is especially necessary in the end-user devices, where the amount of storage space is limited.

4.3.3 How the Nettimer is used in Mobile Device and the Access Node

Using Nettimer in Mobile Devices and the Access Node differs in number of measured connections. In Figure 5 an example architecture is depicted. Seven Mobile Devices have connected to the Access Node. Two of them are connected through a WLAN, three through GPRS and two though GSM network. Access Node’s Nettimer “sees” all the connections, and is able to provide bandwidth information for all the connections. Nettimer on the mobile devices, on the other hand, only measures its own connection to the Access Node.
Figure 5: Using Nettimer in Mobile Devices and the Access Node

It may happen that the Mobile Device connects to the Access Node from NATed network. In this case the Access Node is not able to actively probe the Mobile Device, because Mobile Device’s IP address is not visible to the Access Node. The only IP address the Access Node sees is the address of the NAT box. Knowing that there might be more than one Mobile Device behind the NAT box, the Access Node cannot determine the bandwidth for a specific Mobile Device. If this is the case, the Mobile Device informs the Access Node about the current bandwidth on a regular basis.
5 CONTROL AGENT

Control Agent provides three kinds of services: it can be asked to open/close a communication channel or activate/deactivate a message transport protocol, it may do this autonomously, and it is able to initiate roaming. Each of these services is described in upcoming subsections.

5.1 Request-based operation

Whereas the MA is accessed through MAFunctions-interface, the platform-specific part accesses CA through a CAFunctions-interface, as shown in Figure 6. Behind the CAFunctions-interface there is a set of concrete Function-classes, which are instantiated by CAFunctions, based on the request. All of the Function classes delegate link control functionality to ConnectionManager, which is able to open/close communication channels and activate/deactivate message transport protocols. This kind of design makes the concrete Function classes quite simple, thus it would be tempting to remove the Function class layer altogether and move its functionality to either CAFunctions or ConnectionManager. However, we do not want to rule out the possibility that CA would include functions, which do not need to use ConnectionManager. Moreover, moving the functionality to CAFunctions is not an option, because we want to separate the interface from the concrete implementation.

![Figure 6: Control Agent functionality in request-based operations](image)

The ConnectionManager manages a set of concrete Connector classes. For each message transport connection (MTC) there is a corresponding Connector. In our implementation we are supporting Connectors for WLAN, GSM/HSCSD and GPRS connections, but adding new MTCs is easy by implementing a Connector for it.

Unlike managing MTCs, which is quite a low-level issue, managing the MTPs is dependent on the platform being used. For managing MTPs we have a set of Activator-classes, one for each MTP. These classes are supposed to be able to activate and deactivate a given MTP. However, we only provide implementation for HTTP MTP and our Wireless MTP. For other platforms specific Activator classes must be implemented. As in MTCs, MTPs can be easily added by implementing a new Activator for a corresponding MTP.

In the following, we present a fragment of code taken from the Control Agent. It shows how the CAFunctions-interface is accessed to execute a given CAFunction.
NasOntology ont =
    New NasOntologyFactory ().create (NasOntologyConstants.XML, acl.getContentObject ());
NasOntology returnOnt =
    CAFunctions.getInstance ().executeFunction (ont, parent);
ACL reply = new ACL ();
    reply.setSenderAID (myAid);
    reply.setReceiverAID (acl.getSenderAID ());
    reply.setLanguage (acl.getLanguage ());
    reply.setOntology (acl.getOntology ());
    reply.setProtocol (acl.getProtocol ());
    reply.setInReplyTo (acl.getReplyWith ());
    reply.setConversationID (acl.getConversationID ());
    reply.setContentTypeObject (returnOnt.toXML ());
    reply.setPerformative (FIPACONSTANTS.INFORM);
try
    parent.forward (reply);
catch (Exception e)
    System.err.println ("Sending reply failed: " + e);
    e.printStackTrace ()

At lines 1, 2 and 3 a NasOntology object is created out of the content field of the incoming ACL message. At lines 4 and 5 the newly created NasOntology object is passed to the CAFunctions-interface. CAFunction’s executeFunction method takes care of creating the correct CAFunction and executing it. Once the function is executed, the resulting ontology is returned, which in turn can be sent back to the requesting agent (lines 6-24).

5.2 Requesting CA to open communication channel

Request:

```
[1] public void sendOpenCommChannel ()
[2] {
[3]    ACL acl = new ACL ();
[4]    AgentID target = new AgentID ();
[5]    target.setName ("ca@localap");
[7]    acl.setLanguage (FIPACONSTANTS.CL_REP_XML);
[8]    acl.setReceiverAID (target);
[9]    acl.setPerformative (FIPACONSTANTS.REQUEST);
[10]   CommChannel commChannel = new CommChannel ();
[11]   commChannel.setName ("foo");
[13]   commChannel.addOption ("some", "option");
[14]   OpenCommChannelOntology ont = new OpenCommChannelOntology ();
[15]   ont.setCommChannel (commChannel);
[16]   ont.setAID (myAid);
[17]   ont.setExpression ("action");
[18]   acl.setContentTypeObject (ont.toXML ());
[19]   acl.setSenderAID (myAid);
[20]   forward (acl);
[21] }
```
Reply:

```java
public void handleInform (Conversation conv)
{
    ACL acl = conv.getACL (conv.getLatestMessageIndex ());
    int encoding = XML;
    if (acl.getLanguage () != null) {
        encoding = getContentEncoding (acl.getLanguage ());
    }
    NasOntology ont =
        new NasOntologyFactory ().create (encoding, acl.getContentObject ());
}
```

5.3 Requesting CA to close communication channel

Request:

```java
public void sendCloseCommChannel ()
{
    ACL acl = new ACL ();
    AgentID target = new AgentID ();
    target.setName ("ca@localap");
    target.setAddresses ("("sequence phttp://131.177.251.97/acc")");
    acl.setLanguage (FIPACONSTANTS.CL_REP_XML);
    acl.setReceiverAID (target);
    acl.setPerformative (FIPACONSTANTS.REQUEST);
    CommChannel commChannel = new CommChannel ();
    commChannel.setName (CACONSTANTS.GSM);
    commChannel.setTargetAddr ("phttp://foo.bar:8080/acc");
    commChannel.addOption ("some", "option");
    CloseCommChannelOntology ont = new CloseCommChannelOntology ();
    ont.setCommChannel (commChannel);
    ont.setAID (myAid);
    ont.setExpression ("action");
    acl.setContentObject (ont.toXML ());
    acl.setSenderAID (myAid);
    forward (acl);
}
```

Reply:

```java
public void handleInform (Conversation conv)
{
    ACL acl = conv.getACL (conv.getLatestMessageIndex ());
    int encoding = XML;
    if (acl.getLanguage () != null) {
        encoding = getContentEncoding (acl.getLanguage ());
    }
    NasOntology ont =
        new NasOntologyFactory ().create (encoding, acl.getContentObject ());
}
```
5.4 Requesting CA to activate message transport protocol(s)

Request:

```java
public void sendActivate ()
{
    ACL acl = new ACL ();
    AgentID target = new AgentID ();
    target.setName ("ca-an@localap");
    target.setAddresses ("(sequence phttp://131.177.251.97/acc)");
    acl.setLanguage (FIPACONSTANTS.CL_REP_XML);
    acl.setReceiverAID (target);
    acl.setPerformative (FIPACONSTANTS.REQUEST);
    Vector transports = new Vector ();
    Transport tp1 = new Transport ();
    tp1.setName (CACONSTANTS.GSM);
    tp1.setGwAddr ("phttp://foo.bar/acc");
    tp1.setDestAddr ("phttp://bar.foo/acc");
    Transport tp2 = new Transport ();
    tp2.setName ("bar");
    tp2.setGwAddr ("phttp://foo.bar/acc");
    tp2.setDestAddr ("phttp://bar.foo/acc");
    transports.addElement (tp1);
    transports.addElement (tp2);
    ActivateOntology ont = new ActivateOntology ();
    ont.setTransports (transports);
    ont.setAID (myAid);
    ont.setExpression ("action");
    acl.setContentObject (ont.toXML ());
    acl.setSenderAID (myAid);
    forward (acl);
}
```

Reply:

```java
public void handleInform (Conversation conv)
{
    ACL acl = conv.getACL (conv.getLatestMessageIndex ());
    int encoding = XML;
    if (acl.getLanguage () != null)
    {
        encoding = getContentEncoding (acl.getLanguage ());
    }
    NasOntology ont =
    new NasOntologyFactory ().create (encoding, acl.getContentObject ());
    } catch (Exception e) {
        log.error(e.getMessage ());
    }
```

5.5 Requesting CA to deactivate message transport protocol(s)

Request:

```java
public void sendDeactivate ()
{
    ACL acl = new ACL ();
    AgentID target = new AgentID ();
    target.setName ("ca-an@localap");
    target.setAddresses ("(sequence phttp://131.177.251.97/acc)");
    acl.setLanguage (FIPACONSTANTS.CL_REP_XML);
    acl.setReceiverAID (target);
    acl.setPerformative (FIPACONSTANTS.REQUEST);
    Vector transports = new Vector ();
    Transport tp1 = new Transport ();
    tp1.setName ("foo");
    tp1.setGwAddr ("phttp://foo.bar/acc");
    tp1.setDestAddr ("phttp://bar.foo/acc");
    Transport tp2 = new Transport ();
    tp2.setName ("bar");
    tp2.setGwAddr ("phttp://foo.bar/acc");
    tp2.setDestAddr ("phttp://bar.foo/acc");
    transports.addElement (tp1);
    transports.addElement (tp2);
    DeactivateOntology ont = new DeactivateOntology ();
    ont.setTransports (transports);
    ont.setAID (myAid);
    ont.setExpression ("action");
    acl.setContentObject (ont.toXML ());
    acl.setSenderAID (myAid);
    forward (acl);
}
```

Reply:

```java
public void handleInform (Conversation conv)
{
    ACL acl = conv.getACL (conv.getLatestMessageIndex ());
    int encoding = XML;
    if (acl.getLanguage () != null)
    {
        encoding = getContentEncoding (acl.getLanguage ());
    }
    NasOntology ont =
    new NasOntologyFactory ().create (encoding, acl.getContentObject ());
    forward (acl);
}
```
5.6 Autonomous operation

CA is also able to act autonomously based on the changes in the environment. It is aware of which network interfaces are attached to the device and what their status is. Figure 7 illustrates the autonomous operations.

```
  Network devices are controlled by DeviceHandler, which generates events about the network interface status, for instance when a network interface is attached to the device. We are currently controlling PCMCIA slots and serial port. The event is delivered for all the EventListeners, including CA, that are registered to DeviceHandler. Based on the events, CA can make autonomous decisions on how to react to them. For instance, upon receiving an event of WLAN card insertion at a PCMCIA slot, the obvious reaction from CA would be to check if the WLAN coverage is available, and if it is, to make a connection to the access node.

However, end-users might not be delighted about having an autonomous system making connections on its own. Therefore, the decisions CA makes can be controlled or biased by user preferences. While the user preferences mainly consist of information on how to connect to a certain network and access node, they also dictate the rules on when to prefer a certain connection to another.

5.7 Support for Roaming

Roaming between networks is initiated either by the user or autonomously by the system. In the former case the user replaces the network adapter herself, whereas in the latter case a change in the QoS of the wireless link results in a notification from MA to CA, which in turn may replace the MTP or close the current MTC altogether and open another one, possibly with some other MTP. Therefore, only the latter case can be considered as real seamless roaming while the first one is more like handling a disconnection.

We assume that all the communication is carried out on top of IP. Roaming always includes a re-connection to the access node and it may happen that the mobile device is assigned a different IP address than before. In order to handle IP level roaming, each mobile device is assigned a static and unique mobile device ID, which is advertised to the access node on each connection setup. The Access node handles the mapping between mobile device ID and IP address to correctly identify devices.

When the user removes a network adapter, the CA’s DeviceHandler notifies ConnectionManager about the removal (see Figure 7). ConnectionManager in turn brings down the network interface, and releases any resources associated to it. At this point the communication is suspended and messages are being buffered.

Figure 7: Control Agent’s autonomous operations
As the user reinserts a network adapter, `ConnectionManager` is again notified by the `DeviceHandler`. Based on the network adapter, `ConnectionManager` asks MA for available access networks. If a suitable network is available, the user preferences are checked for valid login parameters for the available access network. The parameters depend on the access network and for instance in the case of GSM network they can be phone number, login ID and password of the Internet service provider. If the parameters are found, a connection is established to the access node and the newly assigned IP address and mobile device ID is advertised. After this, the communication can be resumed and buffers at both mobile device and access node are flushed.

The system may also implicitly initiate roaming when the QoS of the wireless link changes dramatically, for instance so that the throughput drops below a user-defined limit. Should this happen, the CA's `ConnectionManager` receives a notification from MA (see Figure 7). At this point the CA has to make a decision about the possibility for seamless roaming. First of all, there has to be another access network available. Secondly, the mobile device has to have another network adapter inserted, which is able to access the network. Thirdly, the user has to have valid login parameters for the network.

If the CA decides to roam, it first brings up the new network interface and establishes connection to the access node. The messages are then routed through the new network interface. After that the old connection is brought down and any resources associated to it are released.
6 EFFICIENT AGENT COMMUNICATION

Communication is an essential component of the distributed systems, especially in multi-agent systems. To exchange their knowledge, agents should be able to communicate with each other. In wireless environments, the communication should be tailored to provide an efficient use of scarce and fluctuating data communication resources. Furthermore, efficiency is sometimes not so important as, for example, reliability.

Agent communication in wireless environments should be designed for each communication layer, from transport protocol layer up to application layer. Sometimes, applying changes only in one layer might be meaningless as an inefficient solution in another layer deteriorates all the changes in other layers. NAS implementation provides both a message transport protocol designed for wireless environment and an efficient way to encode message envelopes and ACL messages. In the following sections, we describe these in detail.

6.1 Communication architecture in CRUMPET

To support efficient agent communication, NAS implements three main components: Wireless Message Transport Protocol (WMTP), MessageCodec and BufferingService. Figure 8 shows how these components relate to each other. Although in the Figure there are two MTPs, WMTP and IIOP, there could be only WMTP. This is the case when Mobile Devices are used. However, in the Access Node we can have more than one MTP, if interoperability with another platform requires this. WMTP informs ControlAgent, or in fact any entity that has registered with it, about link behaviour, such as disconnections. WMTP is discussed later in more detail.

MessageCodec provides means for encoding/decoding message envelopes and ACL messages. MTPs can make use of this, if needed. MessageCodec and different encodings are discussed later.

BufferingService sits between MTPs and MicroFIPA-OS Multiplex. Its task is to buffer messages in the case of disconnection. BufferingService is controlled by Control Agent, so that based on the current situation, Control Agent may for instance set buffering times or even flush the buffer altogether, if it decides that it is not reasonable to keep messages in the buffer.
6.2 Bit-efficient ACL message

For FIPA-ACL three standard encoding schemes have been specified. The first one is based on ASCII strings, and therefore it is not optimal. The second one is based on XML, which is a highly verbose syntax. The third one is bit-efficient syntax, which is especially designed for wireless environments.

In bit-efficient FIPA-ACL there are two primary ways to reduce the transfer volume over the wireless link: data reduction and intelligent caching. First, FIPA-ACL messages are encoded efficiently by using one-octet codes for predefined message parameters and other common parts of messages. This is a significant improvement compared to a simple string-based coding, as it in most cases reduces additional overhead to half of the original. Furthermore, this improvement is easy to implement and is generally faster to parse than the string-based coding — comparing bytes is much faster than comparing strings. By intelligent caching we mean that similar parts of subsequent messages are not transmitted multiple times over the communication channel, but subsequent occurrences are replaced by short codes. Intelligent caching, however, requires a tight coupling between communicating peers, and thus is inapplicable in some situations. Furthermore, implementation of intelligent caching requires more memory and processing power than a simple data reduction scheme.

The number of bytes needed to represent a FIPA-ACL message is significantly lower than in String and XML-encodings. Using the string encoding we need two times and in XML encoding even three times more bytes than in bit-efficient encoding. In the case of creation and parsing times of a bit efficient FIPA-ACL messages, we are looking at speed-ups of about 30% and 50%, respectively, as compared to String encoding. In Appendix A we have collected some measurements about the efficiency of bit-efficient codecs. For more detailed information about the performance of bit-efficient ACL, see [CIA01].

6.3 Bit-efficient envelope

The purpose of the envelope layer is to enable an independent handling of the transport protocol, for example routing of messages without touching the ACL. FIPA has specified an XML-based syntax [FIPA01b] for the message envelope to complement the XML-based syntax for FIPA-ACL [FIPA01c] and HTTP MTP [FIPA00e]. However, the bit-efficient [FIPA00b] envelope encoding is meant for the wireless environment. This syntax is designed to complement bit-efficient encoding of FIPA-ACL [FIPA00c]. We found out that the size of a message envelope using XML encoding is over twice as large as when using bit-efficient encoding. For more detailed test results, see [CIA01]. Appendix A contains some measurements about the envelope sizes using different encoding schemes.

6.4 MessageCodec

MessageCodec is a supportive class meant to be used by MTPs. It hides the complexity of encoding and decoding message envelopes and ACL messages. It provides a simple interface for both encoding and decoding. It is also easy to extend to support new envelope and/or ACL codecs. In CRUMPET MessageCodec supports the following:

- Bit-efficient envelope
- XML envelope
- String (FIPAOS-specific) envelope
- Java serialized envelope
- Bit-efficient ACL message
- String ACL message
- Java serialized ACL message

6.5 Wireless message transport

Deliverable 3.1 specified that WAP be used in CRUMPET as the protocol over the wireless link. However, we decided not to use WAP for several reasons:

- WAP uses UDP. UDP datagrams can be lost, duplicated, or arrive out-of-order. This means that any application/protocol build on top of UDP, must implement the necessary reliability by itself. WAP adds some reliability on top of UDP, more precisely, with WAP we are able to detect, if the packet did not get delivered to the peer. However, we cannot know if the packet
could not be delivered because of congestion or bit-error and therefore cannot act correctly. If the packet was lost because of congestion, retransmitting the packet immediately is definitely not a good solution; in fact the situation probably gets even worse. However, if the reason were bit-errors in the packet, immediate retransmission would be a good choice. If we set the retransmission timers long, as in WAP, we will lose in performance when there are a lot of packet losses. Thus, the timers should be adjusted according to characteristics of the underlying network.

- WAP, because of UDP, is not able to handle NATed environments. More about NAT later.
- There are only very few of available WAP stacks that could be used in CRUMPET. Furthermore, these are written in C/C++, which might make the porting to other platforms hard.
- Because we prefer MTP written in Java, we would have had to write the WAP stack by ourselves, for which we lacked resources.

Therefore, we came up with another solution. The wireless message transport protocol (WMTP) in CRUMPET is based on Persistent HTTP (P-HTTP) connections. P-HTTP is similar to that of HTTP, but the sender does not close the socket after receiving the reply (see Figure 9), but uses the same socket for subsequent messages. However, for each interaction protocol, two sockets are needed, since P-HTTP allows only sending messaging to one direction over one socket. For example, in the case of FIPA-request protocol, the protocol initiator opens a socket and sends the REQUEST message. After receiving this message, the participant opens a new socket and sends AGREE and INFORM messages using the latter socket. This is illustrated in Figure 10. Other TCP packets than those related to opening a socket and the first FIN packet for each socket are omitted.

Figure 9: Example of packet exchange in HTTP
In WMTP the P-HTTP is optimised in two ways. Firstly, in WMTP only one socket is needed, which is the one the sender opens. The receiving packets are transferred using the same socket. Secondly, WMTP implements reliable message transport and is able to recover from disconnections.

Figure 10: Example of packet exchange in P-HTTP

Figure 11: Example of packet exchange in CRUMPET WMTP
Using only one socket between peers has two benefits. Opening a socket takes time, more specifically one round-trip. In slow wireless networks with long delays this may be over a second. In Figure 21 the performance results of FIPA-Request protocol test with both HTTP and P-HTTP are shown. In the test we measure the time from sending a request to the arrival of inform. Although in P-HTTP there is only socket opened less than in HTTP, the difference in performance can be seen. In Figure 22 the results of the subscription test are shown. In this test, the initiator sends a subscription and receives 15 informs. From the results we can clearly see the effect of overhead caused by opening a socket.

Nowadays, the NAT boxes have gained popularity. A NAT box translates IP addresses so that one NAT box is able to represent a subnetwork with one or more hosts. Addressing a NATed host from outside the NATed subnetwork is usually impossible. Because of this, all the network protocols, which are based on opening socket to both directions in peer-to-peer communication, fail if either or both peers are behind a NAT box. Whereas P-HTTP does not work in NATed environments, WMTP works, when the initiating peer is behind a NAT box. In fact, this is the usual case in CRUMPET, where the Mobile Device may get its IP address from an ISP, who only provides NATed addresses. Therefore, because in CRUMPET the Mobile Device always makes the initial connection to the Access Node, it does not matter if the Mobile Device is behind a NAT box. The only requirement is that the Access Node is not.

Reliability is achieved by using message IDs. In this way WMTP is able to detect if a message is duplicated or if arriving messages are out-of-order. Message IDs are encoded in the HTTP headers.

Appendix B contains performance comparisons between HTTP and WMTP.

To summarize, the benefits of WMTP over WAP are the following:

- WMTP is able to handle NATed environments,
- WMTP is written in pure Java making it easy to port to any Java-enabled platform,
- WMTP (re)uses TCP sockets efficiently,
- WMTP is able to handle disconnections and relies on TCP for other reliability,
- WMTP is a lot more efficient than HTTP and a little more than P-HTTP and only a slightly less efficient than WAP, assuming that in WAP retransmissions do not occur.

In order for Control Agent to be able to manage the behaviour of an MTP the MTP must implement MTPControl interface. The MTPControl interface is as follows:

```java
package microfipaos.mts;

public interface MTPControl {
    /*
     * Returns the unique name of the MTP, for instance "http"
     * for HTTP-MTP
     */
    public String getMTPName ();

    /*
     * Activates the transport.
     */
    public boolean activate ();

    /*
     * Dectivates the transport.
     */
    public boolean deactivate ();

    /*
     * Adds a listener to the transport. Once added, the listener
     * receives notifications about the status changes of the transport,
     * such as notification about disconnection.
     */
    public void addListener (Listener listener);

    /*
     * Removes a listener from the transport.
     */
    public void removeListener (Listener listener);
}
```
Using the MTPControl the ControlAgent is able to activate/deactivate the MTP and set the encoding for message envelopes and ACL messages being sent through the MTP. Using MTPControl interface the Control Agent, or basically any entity, is able to register it as a listener to the status changes occurring in the MTP. To be able to receive status events, the registering entity must implement the MTPStatusListener interface, which is as follows:

```java
package microfipaos.mts;

public interface MTPStatusListener
{
    public void mtpStatusChanged (int mtp, int status);
}
```

It is up to the implementation of the MTP, how and when it informs the registered listeners. In WMTP’s case listeners are informed when a disconnection occurs.
7 DISCONNECTION HANDLING AND BUFFERING

Disconnection handling is an essential operation in wireless environments, where sudden disconnections happen more or less frequently. We are handling disconnections by buffering outgoing messages during a disconnection. As buffering of messages consumes resources, we allow to specify a for buffering time by giving Control Agent an opportunity to set a buffering time parameter to the message envelope. However, the system may override this parameter and empty the buffer before timeout, if the resources on the device are running too low.

Buffering is done by BufferingService, which is located at the MTS level (see Figure 8), more specifically logically below the Multiplexer of the MicroFIPA-OS. When an outgoing message is passed through the Multiplexer, it is assigned a unique message ID. The message along with the message is put into the BufferingService as a BufferItem. Next the message is passed to an MTP for delivery and once the MTP reports (by giving back the message ID) that the message has been delivered successfully, the buffered message is removed from the BufferingService. In the case that the message has not been delivered successfully, the message stays in the BufferingService.

BufferingService is controlled by Control Agent. BufferingService provides an interface, which allows the Control Agent to control the BufferingService, such as flushing the buffered messages, removing them altogether, setting the size of the buffer and the size of the buffering timeouts. The interface is as follows:

```java
package microfipaos.mts;

interface BufferControl
{
    /*
     * Activate buffering, if parameter is "true" and deactivates it if
     * parameter is "false". Once deactivated, BufferingService does not
     * do anything for the messages, but just forwards them.
     */
    public void setBuffering (boolean active);

    /*
     * Returns the current amount of messages in buffer.
     */
    public int getBufferSizeInMessages ();

    /*
     * Returns the current size of the buffer in bytes.
     */
    public int getBufferSizeInBytes ();

    /*
     * Returns the current maximum amount of messages that the buffer can
     * hold.
     */
    public int getMaximumBufferSizeInMessages ();

    /*
     * Sets the maximum size of the buffer in bytes.
     */
    public int setMaximumBufferSizeInBytes ();

    /*
     * Sets the maximum amount of messages that the buffer can
     * hold.
     */
    public int setMaximumBufferSizeInMessages ();
}
```
/*
 * Returns the current maximum size of the buffer in bytes.
 */
public int getMaximumBufferSizeInBytes();

/*
 * Destroys the buffer without trying to send the buffered messages.
 */
public void discardBuffer();

/*
 * First sends the messages and then empties the buffer.
 */
public void flushBuffer();
8 CONFIGURATION FILES

Profiles in NAS contain static information about the parameters for various tasks. All the profiles are stored in ASCII files so that they are easily manageable by a human being. There are four different profiles in NAS: NAS configuration profile, network card profile, access point profile and login info profile. Of these four, NAS configuration profile is encoded in Java Properties-format, whereas the three others are encoded in XML. The reason is that NAS configuration profile is only needed in one device, whereas the other profiles might be shared by a set of agents. Also, writing a parser for Java Properties-formatted profile is trivial as compared to XML parsers. Therefore, these profiles are encoded in a standardized way to promote interoperability. All the profiles are managed by ProfileHandler, which contains parsers for the profiles and provides an interface for accessing the profile information. In the following all the profiles an their entries are described in detail.

8.1 NAS configuration profile

```xml
trace-messages=yes Optional message tracing
access-node=131.177.251.97:10000 MicroFIPAOS-specific access
internal=131.177.251.97:10000 MicroFIPAOS-specific internal
node identifier
address
are-we-access-node=yes Are we access node or client
terminal-id=server@localap Our terminal ID
buffer-size-in-bytes=10000 Buffer size in bytes
buffer-size-in-messages=20 Buffer size in messages
Default-buffer-timeout=5000 Default timeout for implicit
buffer flush
phttp-access-node-host=131.177.251.97 The address where Access Node
is located
phttp-access-node-port=11000 The port that the Access Node
is listening to
```

8.2 Network Card Profile

Network card profile contains information about all the network interfaces that the system supports. An entry in the profile consists of a unique device ID, name, type, linerate, status and a set of device-specific properties. As the Control Agent detects that a network device is attached, it asks (using the unique device ID) the ProfileManager for the profile of the device. Using the profile information it is able to properly initialise the device and check out (from the access point profile), if there are any access points available for this type of device. More about access point profile is described later.

In the following there is an example of network device profile, which contains two entries. The first entry contains profile information of Nokia CardPhone 2.0 GSM modem capable up to 47200 bit/s linerate, whereas the second one contains profile information of Nokia CardPhone 1.0, which only supports 9600 bit/s linerate.

```xml
<?xml version="1.0"?>
<network-devices>
    <network-device>
        <device-id> nokia-cp2 </device-id>
        <name> Nokia Card Phone 2.0 </name>
        <type> gsm </type>
        <linerate> 47200 bit/s </linerate>
        <status> functional.. </status>
        <properties>
            <property>
                <name> init-string </name>
                <value> at+cbst=51,0,1;+chsn=6,0,0,0 </value>
            </property>
        </properties>
    </network-device>
</network-devices>
```
8.3 Access Point Profile

<?xml version="1.0"?>
<access-points>
    <access-point>
        <ap-id> gsm@crumpet.org </ap-id>
        <name> crumpet.org </name>
        <type> gsm </type>
        <owner> sonera </owner>
        <properties>
            <property>
                <name> phone-number </name>
                <value> +35812345678 </value>
            </property>
            <property>
                <name> connection-method </name>
                <value> isdn-120 </value>
            </property>
            <property>
                <name> dhcp </name>
                <value> yes </value>
            </property>
        </properties>
    </access-point>
    <access-point>
        <ap-id> gsm@crumpet.org </ap-id>
        <name> sonera </name>
        <type> gsm </type>
        <owner> sonera </owner>
        <properties>
            <property>
                <name> phone-number </name>
                <value> 0912345678 </value>
            </property>
            <property>
                <name> connection-method </name>
                <value> isdn-120 </value>
            </property>
        </properties>
    </access-point>
</access-points>
<property>
  <name>connection-method</name>
  <value>isdn-120</value>
</property>

<property>
  <name>dhcp</name>
  <value>yes</value>
</property>

</access-points>

8.4 Login Info Profile

<?xml version="1.0"?>
<login-infos>
  <login-info>
    <user-id>laukkmil</user-id>
    <name>Mikko Laukkanen</name>
    <passwd>AdSFSsfdaSDW</passwd>
    <ap-id>gsm@crumpet.org</ap-id>
  </login-info>
  <login-info>
    <user-id>juupajaapa</user-id>
    <name>Juupa Jaapanen</name>
    <passwd>AdSFSsfdaSDW</passwd>
    <ap-id>wlan@crumpet.org</ap-id>
  </login-info>
</login-infos>
9 EXAMPLE USAGE OF NOMADIC APPLICATION SUPPORT

The CRUMPET system builds service adaptation on top of nomadic application support. In CRUMPET the NAS is utilized mainly by three agents: CRUMPET Client Agent (CCA), Dialog Control Agent (DCA) and Content Adaptation Service Agent (CASA). In addition to these, any service agent providing content to the user can benefit from NAS. Moreover, for the autonomous link control, CA in the mobile device subscribes to QoS changes from MA.

CCA resides in the mobile device and handles user interaction. If a user requests explicitly to open/close a communication channel or change the MTP, CCA delegates these to CA. For making autonomous decisions CA uses QoS information provided by MA. DCA is assigned to CCA as the CCA establishes connection to the access node. The main task of DCA is to route user requests from CCA to appropriate service agents. Along with the connection establishment, the CCA sends the terminal type identifier to the DCA. DCA in turn passes this to CASA, which maps the identifier to a terminal profile, which is used as a basis for static content adaptation, which means adaptation to QoS parameters that do not change while the CCA is connected to the access node. CASA is an agent specialized in content adaptation both based on the terminal profile and current QoS of the wireless link that is provided by MA.

In the following we describe a scenario illustrating how the cooperation between these agents enables adapted services for a nomadic user. We begin with an initial situation, where we assume that the system is up and running, but the user has not logged into the system. The scenario is illustrated in Figure 12 and Figure 13, to which the numbers in parentheses in the text refer. Dashed lines in the figures imply ACL communication, whereas solid lines refer to data exchange, which may be applied using non-ACL communication.

9.1 Login procedure

When the user starts the system on her mobile device, the CCA asks CA for a connection to the access node (1). CA then queries MA for a list of available wireless networks (2), and based on the user preferences (3), it establishes a connection to the access node. CA informs CCA about the successful establishment of the connection and CCA may proceed by sending a login request to the DCA (4). The login request contains the device profile identifier, which the DCA forwards to CASA (5). CASA administrates the device profile database (6), and maps the device profile identifiers of the currently logged devices to the actual profiles.

![Figure 12: The login procedure](image-url)
9.2 Accessing adapted services

Once logged in, the user asks CCA for some content, for instance to draw a map of neighboring area with points of interests. CCA forwards the request to DCA (7), which in turn brokers the requests to specific service agents. Service agents, knowing nothing about the current dynamic or the static QoS parameter, reply to DCA with unadapted content (8). It should be noted that in this scenario we do not make any assumption on the content type itself, but assume it to be of discrete type, for instance text, images, video clips etc. DCA uses CASA to adapt the content based on the device profile and current QoS of the wireless link (provided by MA). Based on the static and dynamic QoS parameters CASA decides the appropriate content adaptation algorithm and replies to DCA with adapted content. Finally, DCA replies to CCA, which shows the content to the user.

Figure 13: Accessing adapted services
10 NAS DEPLOYMENT

10.1 Prerequisites
To run NAS implementation in a mobile device, the following prerequisites must be fulfilled:
- Linux distribution (Familiar 0.4 or above) must be installed on the device.
- Java runtime (JDK1.1.x or above, preferably JDK1.1.8) must be available on the device.
- In order to be meaningful the system has to support network interfaces, such as WLAN cards, GSM cards or GPRS cards/phones.

Prerequisites for the Access Node are as follows:
- Access Node must run Linux.
- JDK 1.3 or above must be available on the Access Node.
- Access Node must have at least one network interface and be accessible from access networks, from where the mobile device accesses the CRUMPET system.

10.2 Directory Structure
NAS sources are packaged under package “crumpet.nas”. Figure 14 shows the top level directory structure. The NAS is divided into seven subpackages. Package “be” includes source code for the bit-efficient codecs, both for message envelope and ACL messages. Package “ca” all ControlAgent sources and connection scripts. Package “config” contains sources for reading the configuration files and managing them. Package “ma” includes MonitorAgent sources. Package “ontology” contains both source files for NAS ontology objects and some examples on using them. Package “parser” contains all source code needed for parsing NAS ontology objects in different formats. Package “profile” contains XML-profiles for network devices, access points and login infos. Packages “fipaos” and “microfipaos” include the source code for FIPA-OS and MicroFIPA-OS, respectively. These packages are not implemented nor maintained by us, but package “microfipaos” contains subpackage “microfipaos.mts”, where we have imported our code, namely BufferingService, MessageCodec and wireless message transport protocol.

Figure 14: Top level directory structure of NAS distribution

All these packages are described in more detail in the following sections.
10.3 Package nas.be
Package "be" divides into five subpackages, which include the code needed for bit-efficient codec. In addition to these, the subpackage “examples” contains some code for testing the implementation. Bit-efficient package is shown in Figure 15.

![Figure 15: Directory structure of package "nas.be"](image)

10.4 Package nas.ca
ControlAgent-package is divided into five subpackages, which are shown in Figure 16. Subpackage “connections” contains the code needed for handling connections, such as ConnectionManager and Connector-classes. Subpackage “events” contains the implementation of NAS events, i.e., how the events notifier delivers events to the event listeners. Subpackage “functions” contains classes that implement the request-based operation of Control Agent, that is, the CAFunctions-interface and the concrete CAFunction-classes. Subpackage “pcmcia” contains both Java and C-code for detecting status changes in the PCMCIA slot(s). Subpackage “profile” contains ProfileHandler and the parsers for the XML-profiles (network interfaces, access points and login infos). This subpackage also includes Linux-specific scripts for opening and closing connections.

![Figure 16: Directory structure of package "nas.ca"](image)

10.5 Package nas.ma
This package contains the code for implementing MonitorAgent functionality. It is divided only into two subpackages, “functions” and “gui”. While the “functions” subpackage contains the relevant implementation for the MAFunctions and concrete MAFunction-classes, the “gui” package is an optional subpackage containing a simple implementation for a GUI showing the QoS information.

![Figure 17: Directory structure of package "nas.ma"](image)
10.6 Package nas.ontology
Ontology-package contains implementation of NAS ontology objects. It also contains some examples for using the NAS ontologies. The package is shown in Figure 18.

![Directory structure of package "nas.ontology"](image1)

Figure 18: Directory structure of package "nas.ontology"

10.7 Package nas.parser
Package “parser” contains two kinds of parser implementations: tokenizers and content parsers (see Figure 2). Two kinds of tokenising parsers are implemented, xml and binary xml. In addition to these, Java serialized objects are supported, but as they do not require any external parser, they are not included here. For both xml and binary xml parsers there are some examples of the contents they are able to parse.

All the other parsers are content parsers and they can be identified by their name, which corresponds to the name of the NAS ontology object. For instance, “opencommchannel”-subpackage contains content parser for parsing OpenCommChannelOntology object.

Directory structure of parser classes is shown in Figure 19.

![Directory structure of package "nas.parser"](image2)

Figure 19: Directory structure of package "nas.parser"

10.8 Package nas.trace
NAS trace package is an optional package, which contains code for visualizing agent messaging.

10.9 Packages fipaos and microfipaos
Package “fipaos” and “microfipaos” are imported packages to NAS. They are implemented and maintained by WP2. However, we have done some modifications to “microfipaos”-package, especially the Multiplexer (see Deliverable 2.5) of MicroFIPAOS, so that it is able to utilize the NAS functionality. In addition, our implementation of BufferingService and MessageCodec classes as well as the wireless message transport protocol are bundled under the “microfipaos.mts”-package. These additions and modifications will be included in the final distribution of MicroFIPA-OS.
10.10 Building and installing the distribution

Once the NAS distribution is downloaded and extracted, it can be built using make-tool. The crumpet/src-directory contains top-level Makefile, which can be used for both compiling and installing the NAS distribution. Executing “make all” results in that all the code is compiled to Java class files. After this, executing “make install” produces three Java JAR-files out of the distribution, and they are made available in crumpet/lib-directory. JAR-file named “crumpet.jar” contains crumpet-related classes, including all the NAS classes. JAR-files named “fipaos.jar” and “microfipaos.jar” contain FIPA-OS and MicroFIPA-OS classes, respectively. For more detailed instruction on building the NAS distribution, see file named INSTALL in crumpet-directory.

10.11 Configuring the distribution

NAS can be configured by modifying the profiles using a text editor. To get the NAS up-and-running, the most important profile is the “nas.profile”, but in addition to these, ControlAgent needs information from “nic.profile”, “ap.profile” and “logininfo.profile”. Profiles are discussed in more detail in Section 8.

10.12 Running the distribution

Once the NAS is configured, a platform is NAS-enabled by starting Monitor- and ControlAgent along with the platform startup. Deliverable 2.5 discusses platform startup in more detail, but crumpet/scripts-directory contains example scripts for starting a platform up with Monitor- and Control Agents. For more information about running the NAS agents and NAS-enabling a platform, see file named README in crumpet-directory.

10.13 Maintenance

NAS will be maintained and further developed in CRUMPET project by WP3 until the end of the project. After this, a final release is made and (most of) the code will be made open-source.
REFERENCES


APPENDIX A: EFFICIENCY OF BIT-EFFICIENT CODECS

Table 1: The number of bytes needed to represent ACL message in different encodings

<table>
<thead>
<tr>
<th></th>
<th>Initiator (5 messages)</th>
<th>Participant (8 messages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML</td>
<td>4929</td>
<td>9113</td>
</tr>
<tr>
<td>XML (no content)</td>
<td>2431</td>
<td>3887</td>
</tr>
<tr>
<td>String</td>
<td>3879</td>
<td>7433</td>
</tr>
<tr>
<td>String (no content)</td>
<td>1411</td>
<td>2255</td>
</tr>
<tr>
<td>Bit-efficient</td>
<td>3069</td>
<td>6140</td>
</tr>
<tr>
<td>Bit-efficient (no content)</td>
<td>661</td>
<td>1058</td>
</tr>
<tr>
<td>Bit-efficient (w/ code table)</td>
<td>2661</td>
<td>5431</td>
</tr>
<tr>
<td>Bit-efficient (w/ code table; no content)</td>
<td>253</td>
<td>349</td>
</tr>
</tbody>
</table>

Figure 20: The effect of content payload in different encodings

Table 2: Encoded envelope sizes using different encoding schemes

<table>
<thead>
<tr>
<th></th>
<th>Minimal Envelope</th>
<th>Typical Envelope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-Efficient</td>
<td>151 (100%)</td>
<td>680 (225%)</td>
</tr>
<tr>
<td>IIOP/IDL</td>
<td>339 (225%)</td>
<td>971 (143%)</td>
</tr>
<tr>
<td>XML</td>
<td>585 (387%)</td>
<td>1835 (270%)</td>
</tr>
</tbody>
</table>
APPENDIX B: PERFORMANCE COMPARISON BETWEEN HTTP AND WMTP

Figure 21: Comparison of HTTP and WMTP in the case of FIPA-Request

Figure 22: Comparison of HTTP and WMTP in the case of subscription