Abstract: This Deliverable specifies the nomadic application support in CRUMPET. It contains a requirement analysis, use case models and detailed class designs in UML format. The overall architecture of the nomadic application support in CRUMPET consists of two main entities: the Mobile Device (which in CRUMPET will be a Compaq iPAQ running the LINUX operating system) and the CRUMPET Access Node. The 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. The 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. This Deliverable defines two agents: Monitor Agent and Control Agent. The Monitor Agent monitors the quality of service 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 the QoS changes the subscribed agents are informed. The Control Agent controls the use of the wireless link. It is able to autonomously establish and shutdown connections based on the QoS on the wireless link. The protocol used on the wireless link is WAP as specified by FIPA. The ACL messages inside the WAP packets are encoded bit-efficiently according to the FIPA ACL Message Representation in the Bit-Efficient Specification. In the case of disconnection on the wireless link, the messages are buffered at the CRUMPET Access Node so that when the link is re-established, the buffered messages are transmitted and the session continues.

Keyword List: multi agents, software agents, FIPA, FIPA-OS, nomadic application, WAP
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 in CRUMPET. It contains a requirement analysis, use case models and detailed class designs in UML format.

The overall architecture of the nomadic application support in CRUMPET consists of two main entities: the Mobile Device (which in CRUMPET will be Compaq iPAQ running the LINUX operating system) and the CRUMPET Access Node. These two entities are connected through a wireless link. The 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. The 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 design principle is to follow UML and design patterns as much as possible in order to achieve a modular, extensible and reusable implementation for the nomadic application support. Although the design is targeted to the FIPA-OS agent platform, it does not rule out other agent platforms. The implementation languages used to implement the nomadic application support for CRUMPET are Java and C. Although the Mobile Device and the CRUMPET Access Node in CRUMPET use the LINUX operating system and therefore use a lot of LINUX-specific functionality, the design does not rule out other operating systems.

The design specified in this Deliverable is based on the FIPA Nomadic Application Support specification, which defines two agents: Monitor Agent and Control Agent. The Monitor Agent monitors the quality of service 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 the QoS changes the subscribed agents are informed. The Monitor Agent does a first-level analysis for the collected QoS by calculating statistical quantities from the QoS data.

The Control Agent controls the use of the wireless link. It is able to autonomously establish and shutdown connections based on the QoS on the wireless link. The Control Agent provides an ACL interface for other agents to access its services. It also maintains profiles for both user preferences and network devices. The former one allows users of the wireless link to express their preferences on how to use the link. The latter one describes the devices used to access the wireless networks. The Control Agent in CRUMPET supports GSM, HSCSD, WLAN, GPRS and possibly UMTS.

The protocol used on the wireless link is WAP as specified by FIPA. The WAP message transport is added to FIPA-OS platform for enabling the use of WAP messaging via a WAP gateway. The ACL messages inside the WAP packets are encoded bit-efficiently according to the FIPA ACL Message Representation in the Bit-Efficient Specification. In the case of disconnection on the wireless link, the messages are buffered at the CRUMPET Access Node 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>27.12.00</td>
<td>Initial version.</td>
</tr>
<tr>
<td>0.2</td>
<td>Mikko Laukkanen</td>
<td>27.01.01</td>
<td>First version for comments.</td>
</tr>
<tr>
<td>0.3</td>
<td>Heikki Helin</td>
<td>08.02.01</td>
<td>Comments and inserted BE section.</td>
</tr>
<tr>
<td>0.4</td>
<td>Mikko Laukkanen</td>
<td>16.02.01</td>
<td>Added NAS ontologies.</td>
</tr>
<tr>
<td>0.5</td>
<td>Mikko Laukkanen</td>
<td>02.03.01</td>
<td>Class diagrams and definitions.</td>
</tr>
<tr>
<td>0.6</td>
<td>Mikko Laukkanen</td>
<td>08.03.01</td>
<td>Added CM section.</td>
</tr>
<tr>
<td>0.7</td>
<td>Mikko Laukkanen</td>
<td>15.03.01</td>
<td>Disconnections and buffering section.</td>
</tr>
<tr>
<td>0.8</td>
<td>Mikko Laukkanen</td>
<td>19.03.01</td>
<td>Added Executive Summary, CRUMPET Access Node section and References.</td>
</tr>
<tr>
<td>0.9</td>
<td>Mikko Laukkanen</td>
<td>21.03.01</td>
<td>Fixes according to comments from Sonera's internal review.</td>
</tr>
<tr>
<td>1.0</td>
<td>Mikko Laukkanen</td>
<td>21.03.01</td>
<td>First version for CRUMPET-wide delivery.</td>
</tr>
<tr>
<td>1.1</td>
<td>Mikko Laukkanen</td>
<td>29.03.01</td>
<td>Modifications done according to comments from the WP2/WP3 interim meeting.</td>
</tr>
<tr>
<td>1.2</td>
<td>Mikko Laukkanen</td>
<td>10.04.01</td>
<td>Modifications done according to comments from the CRUMPET partners.</td>
</tr>
<tr>
<td>1.3</td>
<td>Mikko Laukkanen</td>
<td>10.04.01</td>
<td>Final version for the delivery to CEC.</td>
</tr>
<tr>
<td>B1</td>
<td>Martin Potts</td>
<td>14.04.01</td>
<td>Final quality check and sent to the Project Officer.</td>
</tr>
</tbody>
</table>
# ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</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>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>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>HOP</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>JINI</td>
<td>Java Intelligent Network Infrastructure</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</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>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>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>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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Point</td>
<td>Access Point provides wireless access to the CRUMPET Access Node.</td>
</tr>
<tr>
<td>Agent shell</td>
<td>Agent shell is the basic building block of agents in FIPA-OS that provides services such as message passing, tasks and conversations.</td>
</tr>
<tr>
<td>AppAgent</td>
<td>Any CRUMPET agent other than MA or CA.</td>
</tr>
<tr>
<td>Bluetooth</td>
<td>Bluetooth is 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>CA</td>
<td>Control Agent. Controls the MTC and MTP properties according to the FIPA Nomadic Application Support specification.</td>
</tr>
<tr>
<td>Client Agent</td>
<td>CRUMPET User Agent allowing user interface to the CRUMPET services.</td>
</tr>
<tr>
<td>CM</td>
<td>Congestion Manager. Handles the congestion management by controlling the data transfer of TCP and UDP clients. Allows also queries for QoS.</td>
</tr>
<tr>
<td>CRUMPET Access Node</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>CRUMPET Access Node FIPA-OS</td>
<td>Full-blown FIPA-OS supporting also WAP MTP.</td>
</tr>
<tr>
<td>FIPA</td>
<td>Foundation for Intelligent Physical Agents (FIPA) is an international organisation that aims to standardise interoperable software agent architecture.</td>
</tr>
<tr>
<td>FIPA-OS</td>
<td>FIPA-OS is an open-source software agent platform that complies with the FIPA specifications.</td>
</tr>
<tr>
<td>IMTS</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>iPAQ</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>IrDA</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>J2ME</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>J2SE</td>
<td>Java 2 Standard Edition is the traditional Java application environment.</td>
</tr>
<tr>
<td>Kaffe</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>libcm</td>
<td>C-library, which provides an API to access Congestion Manager's (CM) services.</td>
</tr>
<tr>
<td>libcmon</td>
<td>C-library that provides functions for MA (through JNI) to access services provided by CM.</td>
</tr>
<tr>
<td>LINUX</td>
<td>LINUX is a Unix based open source operating system.</td>
</tr>
<tr>
<td>MA</td>
<td>Monitor Agent. Monitors the link properties according to the FIPA Nomadic Application Support specification.</td>
</tr>
<tr>
<td>MicroFIPA-OS</td>
<td>The CRUMPET agent execution environment for small devices that is based on the FIPA-OS agent platform.</td>
</tr>
<tr>
<td>Micro-MTS</td>
<td>Lightweight FIPA-OS MTS designed for small-footprint devices.</td>
</tr>
<tr>
<td>Mobile Device</td>
<td>The 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>Non-agent apps</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>PJava</td>
<td>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.</td>
</tr>
<tr>
<td>Presentation Performer</td>
<td>Presentation Performer takes care of adapting the data coming from fixed network onto the wireless link according to the current QoS of the wireless link.</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation is the Java way of executing distributed remote procedure calls.</td>
</tr>
<tr>
<td>TCP/CM</td>
<td>CM-enabled (LINUX) TCP.</td>
</tr>
<tr>
<td>UDP/CM</td>
<td>CM-enabled (LINUX) UDP.</td>
</tr>
<tr>
<td>WAP Gateway</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 ........................................................................................................................11
2 DESIGN RATIONALE ..................................................................................................................13
3 REQUIREMENTS ..........................................................................................................................14
4 USE CASES ..................................................................................................................................17
  4.1 MOTIVATING SCENARIO .........................................................................................................17
  4.1.1 Scenario .............................................................................................................................17
  4.1.2 Environment .......................................................................................................................17
  4.1.3 Internal functionality ..........................................................................................................18
  4.2 GENERAL USE CASE DIAGRAM ..........................................................................................22
  4.3 USE CASE SCHEMAS ............................................................................................................23
  4.3.1 Monitoring .........................................................................................................................24
  4.3.2 Controlling .........................................................................................................................27
  4.3.3 WAP messaging ..................................................................................................................29
  4.3.4 Disconnection and buffering ..............................................................................................31
5 ARCHITECTURAL OVERVIEW ..................................................................................................33
  5.1 GENERAL NOMADIC ARCHITECTURE ................................................................................33
  5.2 CRUMPET ACCESS NODE ...................................................................................................34
  5.3 CRUMPET AGENT ARCHITECTURE ....................................................................................35
  5.4 MOBILE DEVICE - CRUMPET ACCESS NODE ARCHITECTURE .........................................36
6 NOMADIC APPLICATION SUPPORT ONTOLOGY ..................................................................38
  6.1 FIPA-NAS ONTOLOGY IN CRUMPET ................................................................................38
  6.2 INTERFACES TO OTHER AGENTS .......................................................................................39
  6.3 DESIGN PRINCIPLES ............................................................................................................40
  6.4 CLASS DIAGRAMS ................................................................................................................40
  6.4.1 QosInformationOntology .................................................................................................40
  6.4.2 QosNotificationOntology .................................................................................................41
  6.4.3 OpenCommChannelOntology ..........................................................................................41
  6.4.4 CloseCommChannelOntology .........................................................................................42
  6.4.5 ActivateOntology .............................................................................................................42
  6.4.6 DeactivateOntology .........................................................................................................43
  6.5 CLASS DEFINITIONS ............................................................................................................43
  6.5.1 NasOntologyFactory ..........................................................................................................43
  6.5.2 NasOntology .....................................................................................................................44
  6.5.3 NasParser ..........................................................................................................................44
  6.5.4 SL2Parser ..........................................................................................................................44
  6.5.5 CommChannel ..................................................................................................................45
  6.5.6 TransportProtocol .............................................................................................................45
  6.5.7 NasQos ............................................................................................................................46
  6.5.8 TimeValue ........................................................................................................................46
  6.5.9 RateValue ........................................................................................................................46
  6.5.10 ChangeConstraint ..........................................................................................................46
  6.5.11 TimeConstraint ...............................................................................................................47
  6.5.12 QosInformationOntology ..............................................................................................47
  6.5.13 QosInformationParser ....................................................................................................48
  6.5.14 QosNotificationOntology ...............................................................................................48
  6.5.15 QosNotificationParser ....................................................................................................49
  6.5.16 OpenCommChannelOntology .......................................................................................49
  6.5.17 OpenCommChannelParser ...........................................................................................50
  6.5.18 CloseCommChannelOntology .......................................................................................50
  6.5.19 CloseCommChannelParser ...........................................................................................51
  6.5.20 ActivateOntology Parser ..............................................................................................51
9 CONGESTION MANAGER ........................................................................................................ 96
  9.1 DESCRIPTION ........................................................................................................................ 96
  9.2 CONSTRAINTS AND LIMITATIONS ......................................................................................... 98
  9.3 INTERFACES .......................................................................................................................... 98
10 WAP TRANSPORT .................................................................................................................. 100
  10.1 DESCRIPTION ....................................................................................................................... 100
  10.2 CONSTRAINTS AND LIMITATIONS ......................................................................................... 100
  10.3 ARCHITECTURE ...................................................................................................................... 100
  10.4 INTERFACES .......................................................................................................................... 102
      10.4.1 Mobile device side ........................................................................................................... 102
      10.4.2 WAP Gateway domain .................................................................................................. 105
      10.4.3 CRUMPET Access Node side .......................................................................................... 105
  10.5 CLASS DIAGRAMS ............................................................................................................... 106
      10.5.1 Mobile Device side ........................................................................................................ 106
      10.5.2 CRUMPET Access Node side ........................................................................................ 107
  10.6 CLASS AND MODULES DEFINITIONS ............................................................................. 107
      10.6.1 WAPComms ................................................................................................................... 107
      10.6.2 HTTPSender .................................................................................................................. 108
      10.6.3 HTTPReceiver ............................................................................................................... 109
11 BIT-EFFICIENT MESSAGING SUPPORT ........................................................................... 110
  11.1 DESCRIPTION ....................................................................................................................... 110
  11.2 CONSTRAINTS AND LIMITATIONS ....................................................................................... 111
  11.3 INTERFACES ........................................................................................................................ 111
  11.4 CLASS DIAGRAMS ............................................................................................................... 112
  11.5 CLASS DEFINITIONS .......................................................................................................... 113
      11.5.1 ACLConstants ............................................................................................................... 113
      11.5.2 ACLEncoder ................................................................................................................. 114
      11.5.3 ACLInputStream ......................................................................................................... 114
      11.5.4 ACLOutputStream ..................................................................................................... 115
      11.5.5 BinDate ......................................................................................................................... 115
      11.5.6 BINumber ..................................................................................................................... 116
      11.5.7 BinRep ............................................................................................................................ 116
      11.5.8 DecoderCodetable ....................................................................................................... 117
      11.5.9 EncoderCodetable ....................................................................................................... 118
      11.5.10 EnvelopeConstants ..................................................................................................... 118
      11.5.11 EnvelopeEncoder ........................................................................................................ 119
      11.5.12 EnvelopeDecoder ....................................................................................................... 119
  11.6 BIT-EFFICIENT CONTENT LANGUAGE ........................................................................ 119
12 DISCONNECTIONS AND BUFFERING .............................................................................. 120
12.1 DESCRIPTION.........................................................................................................................120
12.2 BUFFERING PARAMETERS....................................................................................................121
12.3 MOBILE-TERMINATED MESSAGE BUFFERING .................................................................121
12.4 MOBILE-ORIGINATED MESSAGE BUFFERING .................................................................122
12.5 CONSTRAINTS AND LIMITATIONS .....................................................................................122
12.6 INTERFACES ......................................................................................................................122
12.7 CLASS DIAGRAM.................................................................................................................122
12.8 CLASS DEFINITIONS ...........................................................................................................123
   12.8.1 MessageBufferService.............................................................................................123
   12.8.2 BufferRouter ...........................................................................................................124
   12.8.3 Buffer .....................................................................................................................125
13 REFERENCES ........................................................................................................................126
1 INTRODUCTION

The overall aim of the CRUMPET project [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 the 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. The implementation will be developed using best practice, including “open source for open standards” development, developing generic re-usable components on an open distributed architecture.

The adaptation of tourist services to different nomadic computing and network environments and varying network conditions is an important area. There are several tasks that intelligent agents need to carry out during application adaptation:

- Selection of Message Transport Protocol and Message Transport Connection to be used for the agent communication.
- Provision of support for application agents to carry out the adaptation of application data, such as still images, video and audio, XML, etc. Today’s Internet application data (such as multimedia content) is designed with high performance desktop PCs and high quality displays in mind. Therefore, the application data is frequently unsuitable for nomadic computing using wireless wide-area networks and low-performance Mobile Devices, and hence requires modification or adaptation.
- Communication between the agents performing the adaptation.

The agent middleware for nomadic application support in CRUMPET consists of the following logical agents:

1. Monitor Agent, which carries out the tasks of observing the QoS of the Message Transport Protocol (MTP) and the Message Transport Connection (MTC), measuring (if there are no other means to obtain the required information) the QoS of the MTP and the MTC, collecting information from the observing and measuring sources, analysing the information,

2. Control Agent, which carries out the task of controlling and selecting the MTC and MTP.

The most appropriate configuration of the Monitor Agent and Control Agent is that there is at least one pair in each host of the Agent Platform involving adaptation. The Monitor Agent may measure the actual QoS of the MTC, if the network running the MTC does not provide users with required performance data. The Monitor Agent may provide a repository for the measurement data collected. The Monitor Agent may do the first level analysis of the collected data. The Monitor Agent then sends the results of the analysis to the Control Agent, if requested to do so. The Control Agent may activate the establishment, closing, suspension, activation, etc. of the Message Transport Connection. In some cases there is a need for the Monitor Agents and Control Agents in heterogeneous Agent Platforms to communicate with each other (for example, to achieve a common understanding about the current situation).

The architecture illustrates an agent-based distributed system that offers its services at the best obtainable QoS in a wide variety of environments. A possible agent architecture is illustrated in Figure 1. There are three separate agent platforms:

1. Mobile Host,
2. Access Node, and
3. Fixed Network Host.

The platform agents are the following. The Control Agent activates the establishment, disconnection, suspension, activation, etc. of the connection between platforms. The Control Agent also informs
application agents (on request) about the status and changes of the network services. The Monitor Agent also informs application agents (on request) about the status of the communications. The Mobile Host and the Access Node are connected by a wireless link using (for example) WAP, whereas the Access Node and the Fixed Network Host are connected by a fixed connection and use (for example) IIOP or any other FIPA-specified message transport protocol.

The objective of this document is to specify the design for the agents and other components. WP3 is responsible for the implementation in CRUMPET. This basically means, but is not limited to, the design of the nomadic application specification [FIPA00a] specified by FIPA.

This document is organised as follows. Section 2 defines the design rationale for the nomadic application support in CRUMPET. Section 3 defines the requirements, which the nomadic application support in CRUMPET should fulfil. Section 4 introduces the use cases, which serve as a basis for the design. First there is an overall and motivating scenario, which is then broken down into smaller use cases. Section 5 depicts the architecture of the nomadic application support and therefore concentrates on the (wireless) interface between the terminal device and the CRUMPET gateway. Sections 6, 7, 8, 9, 10, and 11 are the actual design sections presenting the detailed designs for nomadic application support ontologies, Monitor Agent, Control Agent, the exploitation of the congestion manager in CRUMPET, the design of the WAP transport, and the use of bit-efficient encoding, respectively. Section 12 defines, how disconnections are supported in CRUMPET.
2 DESIGN RATIONALE

Current tourism services for nomadic users are in their infancy. Mobile phone users can get the same voice services as they can from a fixed phone and text-based travel information is also available. Internet users have a much richer supply of tourism information and services, although even this is in its infancy compared to what will be possible once e-business establishes itself. Customers are increasingly using the Internet to book holidays, make travel arrangements and to find out information about their destination. Mobile services, such as WAP [WAP01a] and GPRS [GPRS01], enable nomadic use of these applications and offer opportunities for new ones, a trend that will accelerate with third-generation mobile networks. Some of the current limitations of Internet tourism services are: knowing where to find the information, extracting the appropriate information, and receiving the information in a form that is easy to digest and suitable to the user context, the transport characteristics of the delivery network, and the terminal capabilities. A key limitation of Internet services that is starting to be overcome with the introduction of WAP is the accessibility to nomadic users. The introduction of WAP technology will not automatically enable access to the vast range of Internet based tourism services currently available, but will simply enable Internet services to be adapted for delivery to WAP enabled devices. CRUMPET is developing a service environment that will generically integrate the traditional Internet and wireless services.
3 REQUIREMENTS

In this section the requirements for the nomadic application support in CRUMPET are specified. All the requirements are organised into tables, each of which encompasses one entity in the whole architecture. The Mobile Device table (Table 1) defines the requirements for the Mobile Device used in CRUMPET. The Monitor Agent (Table 2) and Control Agent (Table 3) tables define the requirements for the FIPA-specified monitor and control agents, respectively. The Bit-efficient communication (Table 4) and WAP transport (Table 5) tables define the requirements in terms of communication over the wireless link. The CRUMPET Access Node table (Table 6) defines the requirements for the entity mediating between the Mobile Device and the CRUMPET services in the fixed network. The Profile requirements (Table 7) are for the persistent storage of the MA and the CA.

Each of the requirements in the tables contain:

- A reference number, which is used to uniquely identify the requirement
- A description of the requirement
- A category, which can be either mandatory (stating that the requirement must be fulfilled) or optional (stating that the requirement is not crucial for the functionality).

Correct and thorough requirements specifications are essential to a successful project. Requirements are a description of the needs or desires for the system. The primary goal of the requirements phase is to identify and document what is really needed, in a form that clearly communicates to the client and to the development team members [Lar98].

It must be noted that the requirements must be measurable and unambiguous, meaning that when comparing the implementation against the requirements, it must be trivial to classify the requirement as fulfilled or not fulfilled. Thus, the requirements can be used as a basis for the user acceptance testing.

<table>
<thead>
<tr>
<th>REF #</th>
<th>Description of Requirement</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Mobile Device</td>
<td></td>
</tr>
<tr>
<td>R1.1</td>
<td>Mobile Device is able to run LINUX.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R1.2</td>
<td>Mobile Device is capable of having MicroFIPA-OS installed on RAM.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R1.3</td>
<td>Mobile Device is capable of having MicroFIPA-OS installed on Flash memory.</td>
<td>Optional</td>
</tr>
<tr>
<td>R1.4</td>
<td>Mobile Device has at least one PC Card slot.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R1.5</td>
<td>Mobile Device has a serial cable connector.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R1.6</td>
<td>Mobile Device and its operating system are able to run Java.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R1.7</td>
<td>Mobile Device and its operating system are capable of interfacing WLAN.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R1.8</td>
<td>Mobile Device and its operating system are capable of interfacing HSCSD.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R1.9</td>
<td>Mobile Device and its operating system are capable of interfacing GPRS.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R1.10</td>
<td>Mobile Device and its operating system are capable of interfacing UMTS.</td>
<td>Optional</td>
</tr>
</tbody>
</table>
Table 2: Monitor Agent requirements.

<table>
<thead>
<tr>
<th>REF #</th>
<th>Description of Requirement</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>Monitor Agent (MA)</td>
<td></td>
</tr>
<tr>
<td>R2.1</td>
<td>MA is able to answer to ACL queries about current QoS.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R2.2</td>
<td>MA is able to take in subscriptions from other agents about changes in the QoS.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R2.3</td>
<td>MA is able to maintain several subscriptions at the same time.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R2.4</td>
<td>MA registers with DF.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R2.5</td>
<td>MA monitors QoS of communication channel</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R2.6</td>
<td>MA monitors QoS of transport protocol.</td>
<td>Optional</td>
</tr>
<tr>
<td>R2.7</td>
<td>MA wraps third-party software in measuring the QoS.</td>
<td>Optional</td>
</tr>
<tr>
<td>R2.8</td>
<td>MA measures throughput.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R2.9</td>
<td>MA measures round-trip time.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R2.10</td>
<td>MA is aware of the linerate of MTC.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R2.11</td>
<td>MA measures all the QoS parameters as specified in the FIPA Nomadic Application Support specification [FIPA00a].</td>
<td>Optional</td>
</tr>
<tr>
<td>R2.12</td>
<td>MA is aware of available network devices and network access.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R2.13</td>
<td>MA is aware of current status of the network connections.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R2.14</td>
<td>MA has a GUI.</td>
<td>Optional</td>
</tr>
<tr>
<td>R2.15</td>
<td>MA uses Fipa-Nomadic-Application ontology.</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

Table 3: Control Agent requirements.

<table>
<thead>
<tr>
<th>REF #</th>
<th>Description of Requirement</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>Control Agent (CA)</td>
<td></td>
</tr>
<tr>
<td>R3.1</td>
<td>CA is able to activate a transport on request.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R3.2</td>
<td>CA is able to deactivate a transport on request.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R3.3</td>
<td>CA is able to open a communication channel on request.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R3.4</td>
<td>CA is able to close a communication channel on request.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R3.5</td>
<td>CA is able to autonomously activate a transport.</td>
<td>Optional</td>
</tr>
<tr>
<td>R3.6</td>
<td>CA is able to autonomously deactivate a transport.</td>
<td>Optional</td>
</tr>
<tr>
<td>R3.7</td>
<td>CA is able to autonomously open a communication channel.</td>
<td>Optional</td>
</tr>
<tr>
<td>R3.8</td>
<td>CA is able to autonomously close a communication channel.</td>
<td>Optional</td>
</tr>
<tr>
<td>R3.9</td>
<td>CA is able to negotiate the transport protocol to be used.</td>
<td>Optional</td>
</tr>
<tr>
<td>R3.10</td>
<td>CA registers with DF.</td>
<td>Optional</td>
</tr>
<tr>
<td>R3.11</td>
<td>CA maintains transport profiles and user preferences, which contain transport-specific parameters.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R3.12</td>
<td>CA uses Fipa-Nomadic-Application ontology.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R3.13</td>
<td>CA has a GUI.</td>
<td>Optional</td>
</tr>
<tr>
<td>R3.14</td>
<td>CA is aware of QoS.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

Table 4: Bit-efficient communication requirements.

<table>
<thead>
<tr>
<th>REF #</th>
<th>Description of Requirement</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>R4</td>
<td>Bit-efficient (BE) communication</td>
<td></td>
</tr>
<tr>
<td>R4.1</td>
<td>BE is used on wireless links.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R4.2</td>
<td>BE is used on wireline links.</td>
<td>Optional</td>
</tr>
<tr>
<td>R4.3</td>
<td>BE uses codetables.</td>
<td>Optional</td>
</tr>
<tr>
<td>R4.4</td>
<td>BE envelope [FIPA00b] is used whenever BE ACL [FIPA00c] is used.</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>
Table 5: WAP transport requirements.

<table>
<thead>
<tr>
<th>REF #</th>
<th>Description of Requirement</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>R5</td>
<td>WAP transport</td>
<td></td>
</tr>
<tr>
<td>R5.1</td>
<td>WAP gateway is separate from FIPA-OS [Pos99].</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R5.2</td>
<td>Existing WAP gateways are used.</td>
<td>Optional</td>
</tr>
<tr>
<td>R5.3</td>
<td>Existing WAP clients are exploited in the Mobile Devices.</td>
<td>Optional</td>
</tr>
<tr>
<td>R5.4</td>
<td>WAP client is implemented in Java.</td>
<td>Optional</td>
</tr>
<tr>
<td>R5.5</td>
<td>POST [WAP00a] method is used in mobile-originated communication.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R5.6</td>
<td>PUSH [WAP01b] method is used in mobile-terminated communication.</td>
<td>Optional</td>
</tr>
<tr>
<td>R5.7</td>
<td>WAP transport is able to recover from disconnections.</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

Table 6: CRUMPET Access Node requirements.

<table>
<thead>
<tr>
<th>REF #</th>
<th>Description of Requirement</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>R6</td>
<td>CRUMPET Access Node</td>
<td></td>
</tr>
<tr>
<td>R6.1</td>
<td>CRUMPET Access Node implements required WAP gateway</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>functionalities.</td>
<td></td>
</tr>
<tr>
<td>R6.2</td>
<td>CRUMPET Access Node FIPA-OS has IIOP [FIPA01a] interface.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R6.3</td>
<td>CRUMPET Access Node FIPA-OS has WAP [FIPA00d] interface.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R6.4</td>
<td>CRUMPET Access Node FIPA-OS has proprietary interfaces to the fixed and wireless network.</td>
<td>Optional</td>
</tr>
<tr>
<td>R6.5</td>
<td>WAP gateway and CRUMPET Access Node FIPA-OS use HTTP [FIPA00e] in their communication.</td>
<td>Optional</td>
</tr>
<tr>
<td>R6.6</td>
<td>CRUMPET Access Node FIPA-OS supports disconnection.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R6.7</td>
<td>CRUMPET Access Node FIPA-OS supports buffering of the ACL messages.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R6.8</td>
<td>CRUMPET Access Node FIPA-OS supports redirection of the ACL messages.</td>
<td>Optional</td>
</tr>
<tr>
<td>R6.9</td>
<td>CRUMPET Access Node runs LINUX as its operating system.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R6.10</td>
<td>CRUMPET Access Node is able to run FIPA-OS.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R6.11</td>
<td>CRUMPET Access Node supports switching from one MTC to another.</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

Table 7: Profile requirements.

<table>
<thead>
<tr>
<th>REF #</th>
<th>Description of Requirement</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7</td>
<td>Profile requirements</td>
<td></td>
</tr>
<tr>
<td>R7.1</td>
<td>CA stores network device properties to a persistent storage.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R7.2</td>
<td>CA stores user preferences to a persistent storage.</td>
<td>Mandatory</td>
</tr>
<tr>
<td>R7.3</td>
<td>MA stores QoS value history to a persistent storage.</td>
<td>Optional</td>
</tr>
</tbody>
</table>
4 USE CASES

4.1 Motivating Scenario

4.1.1 Scenario

The motivating scenario for the design of nomadic application support in CRUMPET is presented below. This scenario is the same as in Deliverable 1.1 [CRU01b] and is a refined version of the scenario presented in the technical annex [CRU00a]. The entities and agents in the scenario are based on the architecture described in Deliverable 1.1 [CRU01b], but are defined in the glossary and further inspected in the upcoming sections. Deliverable 1.1 does not state how to implement the entities as agents. This document, on the other hand, specifies the design based on agent technology. Therefore, we have mapped the entities in the Deliverable to agents as presented in Table 8.

<table>
<thead>
<tr>
<th>Entity in Deliverable 1.1</th>
<th>Mapped agent name in this document</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRUMPET (Client)</td>
<td>ClientAgent</td>
</tr>
<tr>
<td>QoSMonitor in PDA</td>
<td>MonitorAgent in PDA</td>
</tr>
<tr>
<td>QoSControl in PDA</td>
<td>ControlAgent in PDA</td>
</tr>
<tr>
<td>DialogControl</td>
<td>DialogControl</td>
</tr>
<tr>
<td>UserModeling</td>
<td>UserModeling</td>
</tr>
<tr>
<td>Broker</td>
<td>BrokerAgent</td>
</tr>
<tr>
<td>ContentInterfaceLocalEvents</td>
<td>ContentInterfaceLocalEvents</td>
</tr>
<tr>
<td>LocalEventsService</td>
<td>LocalEventsService</td>
</tr>
<tr>
<td>PresentationPerformer</td>
<td>PresentationPerformer</td>
</tr>
</tbody>
</table>

The scenario is as follows:

"Jonathan, as a member of EU inspection committee, is visiting Helsinki to carry out the annual inspection of Finland's special requirements for EU support. The meeting ended earlier than expected, and he has an afternoon and evening off. He is keen to see a local attraction. Jonathan uses his PDA and activates the CRUMPET system to search for interesting things to do. The system accesses the Internet via the wireless LAN provided by the hotel Jonathan is staying in. CRUMPET uses knowledge of Jonathan's preferences and current location. It finds information about a show from the tourist service of Helsinki tourist office that might interest Jonathan. Jonathan wants to see more information about this show. The system begins to display video extracts of the show. Jonathan finds the show interesting and prepares to leave for the evening show.

While waiting for a taxi, Jonathan continues to look at the video clips of the show from his PDA. Jonathan leaves the hotel building, and his PDA seamlessly roams from the hotel's wireless LAN to a public mobile network. There is insufficient bandwidth for the full video extract to be transmitted to the PDA, and the system adapts the data so that only voice is transmitted over the mobile network, and Jonathan continues to listen to soundtrack music of the show."

4.1.2 Environment

The mobile equipment is either a high-end PDA or a low-end laptop, which is easy to carry and use when moving. The mobile equipment needs to have interfaces to the necessary wireless networks. The interface to the wireless LAN is a WLAN (802.11b) PC Card and to the public mobile network, such as High Speed GSM, GPRS [GPRS01], and UMTS [UMTS01], is either through a PC Card modem or through Bluetooth, IrDA [IrDA00] or serial cable via a mobile phone. There are two kinds of data transmission over wireless links: messaging and file transfer (video clips). The amount of data transmitted over wireless links is large.
4.1.3 Internal functionality

“Jonathan, as a member of EU inspection committee, is visiting Helsinki to carry out the annual inspection of Finland’s special requirements for EU support. The meeting ended earlier than expected, and he has an afternoon and evening off. He is keen to see a local attraction. Jonathan uses his PDA and activates the CRUMPET system to search for interesting things to do.”

**Prerequisite:** The Mobile Device is started up with WLAN PC Card and Bluetooth, IrDa or serial cable connection to Jonathan’s mobile phone. (e.g. GPRS / UMTS access could be via Bluetooth, infrared or serial cable).

1. User starts up the CRUMPET and activates ClientAgent.
2. The startup procedure starts up MonitorAgent in PDA.
3. The startup procedure starts up ControlAgent in PDA.
4. MonitorAgent in PDA senses wireless networks operating at the hotel and finds both WLAN and GPRS.
5. ControlAgent in PDA asks MonitorAgent in PDA for currently available wireless networks.
6. MonitorAgent in PDA informs ControlAgent in PDA about the available wireless networks (WLAN and GPRS).
7. ControlAgent checks User’s preferences and finds out that WLAN and GPRS are user-preferred networks and User prefers WLAN over GPRS.
8. ControlAgent in PDA subscribes to information about changes in QoS of WLAN and GPRS from MonitorAgent in PDA.
9. ControlAgent in PDA activates WLAN based on Jonathan’s preferences.
10. (ControlAgent does login on behalf of Jonathan to access hotel’s WLAN, if required.)
"The system accesses the Internet via the wireless LAN provided by the hotel Jonathan is staying in. CRUMPET uses knowledge of Jonathan's preferences and current location. It finds information about a show from the tourist service of Helsinki tourist office that might interest Jonathan"

**Prerequisite:** DialogueControl is registered with the DirectoryFacilitator in the CRUMPET Access Node FIPA-OS.

1. ClientAgent locates the DialogueControl by querying its address from the DirectoryFacilitator in the CRUMPET Access Node FIPA-OS.
2. ClientAgent requests DialogueControl for current interesting attractions in Helsinki.
3. DialogueControl requests the user preferences from UserModelingAgent.
4. DialogueControl constructs a request based on the user preferences.
5. DialogueControl sends the request to the BrokerAgent.
6. BrokerAgent finds out which attractions are of interest to the user.
7. After tracking the interesting attractions BrokerAgent responds to DialogueControl with the list of attractions.
8. DialogueControl forwards the response to the PresentationPerformer.
9. PresentationPerformer forwards the response to the ClientAgent
10. ClientAgent shows the list of attractions to the user, who decides to see more information about a show-taking place in Helsinki.

"Jonathan wants to see more information on this show. The system begins to display video extracts of the show. Jonathan finds the show interesting and prepares to leave for the evening show."
ClientAgent creates a video output interface to show the video clips from the show.

ClientAgent subscribes to QoS information (including significant changes) from MonitorAgent in PDA.

MonitorAgent in PDA queries current throughput (inbound).

MonitorAgent in PDA informs ClientAgent that the throughput (inbound) is above 2 Mbits/sec, which is adequate for retrieving whole video clips.

ClientAgent sends a request to PresentationPerformer to perform content adaptation for the data coming from the LocalEventService.

PresentationPerformer subscribes to QoS information from MonitorAgent in CRUMPET Access Node FIPA-OS.

PresentationPerformer queries current throughput (outbound).

MonitorAgent in CRUMPET Access Node FIPA-OS informs PresentationPerformer that the throughput (outbound) is above 2 Mbits/sec, which is adequate for retrieving the whole video clip.

ClientAgent and LocalEventService (via BrokerAgent and LocalContentInterface) negotiate the format of the video clips.

ClientAgent requests ControlAgent in PDA to open a new ‘transport’ connection over the WLAN for the video stream.

ControlAgent in PDA opens a new transport connection between the PDA and CRUMPET Access Node.

ClientAgent sends a request to LocalEventService (via Broker and LocalContentInterface) to retrieve the video clips.

LocalEventService begins to transmit the video clips (via LocalContentInterface, BrokerAgent and PresentationPerformer) to ClientAgent (and its video output interface).
"While waiting for a taxi, Jonathan continues to look at the video clips of the show from his PDA. Jonathan leaves the hotel building, and his PDA seamlessly roams from the hotel's wireless LAN to a public mobile network."

[1] MonitorAgents in PDA and in the CRUMPET Access Node FIPA-OS sense that either the QoS of WLAN drops rapidly (throughput has dropped below a predefined limit) or the WLAN link is disconnected.


[4] (ClientAgent may ask the ControlAgent to disconnect from WLAN).

[5] PresentationPerformer suspends the transmission of the video clips and begins to buffer the data.

[6] MonitorAgent in PDA senses currently available wireless networks and finds GPRS.

[7] MonitorAgent in PDA informs ControlAgent in PDA that GPRS is available.

[8] ControlAgent in PDA activates GPRS connection (based on the user profile)
   ⇒ Through Bluetooth, IrDA or serial cable to the GPRS mobile terminal.
   ⇒ (fipa.mts.mtp.wap.std message transport protocol is activated.)

[9] MonitorAgent in PDA informs the ClientAgent about the QoS of the GPRS connection.

"There is insufficient bandwidth for the full video extract to be transmitted to the PDA, and the system adapts the data so that only voice is transmitted\(^1\) over the mobile network, and Jonathan continues to listen to soundtrack music of the show".

**4.2 General use case diagram**

Based on the motivating scenario, four packages can be identified:

- Monitoring tasks, which include functionality to monitor the underlying networks and their QoS.
- Controlling tasks, which include functionality to control the communication links.
- Disconnection and buffering, which includes the functionality to handle disconnections and message buffering and redirection.
- WAP messaging, which defines how the WAP transport works.

\(^1\) This is only an example of one kind of content adaptation, which could be applied on the data presented in the scenario. It should be noted that this kind of content adaptation may not be suitable for instance in situations, where the other parts of the content are dependent on the audio, or vice versa.
In Figure 2 these packages are shown with the relations to each other. Associations are depicted in dashed lines. Later, in the upcoming subsections, all the packages and use cases inside them are described in more detail.

![Diagram of use case packages]

**Figure 2: General use case packages.**

4.3 **Use case schemas**

This subsection is further divided into subsections, in which the packages described in the previous section and the use cases they contain are described in detail. The idea is to break down the general functionality into small and easily administered use cases, which can be used later on as a basis for the functional test cases. Also, as the use cases generally outline the course of events as well as define, what happens and what is needed in various situations, they serve as an input to the designs of the agents and components, which specify, how things happen in the situations defined by the use cases.

Actors in UML methodology are external entities participating in the story of the use case either giving some input to - or getting some output from - the system [Lar98]. Usually actors are users of the system and in UML; there is a clear distinction between the actor of the system and the system itself. Now, when talking about autonomous agents, the situation is a little more problematic, since the agents themselves are usually both the stimulators in the use case and the actual entities implementing the use case. We have excluded the real user from the use cases for purpose, because in our opinion the autonomous agents should be the ones who take part in the use cases.
4.3.1 Monitoring

Figure 3: Use cases in monitoring package.

<table>
<thead>
<tr>
<th>Use Case:</th>
<th>1.1 Sense Networks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors:</strong></td>
<td>MA@Mobile Device, MA@CRUMPET Access Node FIPA-OS</td>
</tr>
<tr>
<td><strong>Purposes:</strong></td>
<td>This use case shows how the Monitor Agent senses the currently available networks.</td>
</tr>
<tr>
<td><strong>Cross References:</strong></td>
<td>R1.7, R1.8, R1.9, R1.10, R1.11, R2.5, R2.12, R2.13</td>
</tr>
</tbody>
</table>

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. MA receives either external or internal event for sensing the currently available networks.</td>
<td>-</td>
</tr>
<tr>
<td>2. MA checks out currently connected network devices.</td>
<td>A list of currently connected network devices is constructed.</td>
</tr>
<tr>
<td>3. For every device in the list, MA asks the device for its current status.</td>
<td>A list of mappings between devices and their status is constructed.</td>
</tr>
<tr>
<td>4. MA updates its internal knowledge of the available networks.</td>
<td>MA has the knowledge of currently available networks.</td>
</tr>
</tbody>
</table>
Use Case: **1.2 Subscription Matching**

**Actors:** MA@Mobile Device, MA@CRUMPET Access Node FIPA-OS

**Purposes:** This use case outlines how subscriptions are matched.

**Cross References:** R2.2, R2.3, R2.4

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. MA's QoS subscription-matching component asks the MA for current QoS values.</td>
<td>Current QoS values are received.</td>
</tr>
<tr>
<td>2. For every subscription, matching between the QoS values and the subscription's QoS attributes and constraints is applied.</td>
<td>A list of agents with matching subscription is constructed.</td>
</tr>
<tr>
<td>3. For every matched agent, an inform-ACL message is constructed and sent.</td>
<td>Every matched agent receives the changed QoS information.</td>
</tr>
</tbody>
</table>

Use Case: **1.3 Measure/calculate QoS**

**Actors:** MA@Mobile Device, MA@CRUMPET Access Node FIPA-OS

**Purposes:** This use case outlines how the current QoS is measured/calculated.

**Cross References:** R2.1, R2.7, R2.8, R2.9, R2.10, R2.11

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. MA's QoS data-collecting component iterates through all the QoS measuring functions and receives the measured/calculated values.</td>
<td>Current QoS values are received.</td>
</tr>
<tr>
<td>2. All the QoS values are put into a structure.</td>
<td>QoS structure is created.</td>
</tr>
<tr>
<td>3. MA is updated with the new QoS values.</td>
<td>MA has gained knowledge about the current QoS values.</td>
</tr>
</tbody>
</table>

Use Case: **1.4 Get Access Methods**

**Actors:** MA@Mobile Device, MA@CRUMPET Access Node FIPA-OS, ClientAgent@Mobile Device, PresentationPerformer@CRUMPET Access Node FIPA-OS

**Purposes:** This use case shows, how MA can be asked for currently available access methods.

**Cross References:** R2.12, R2.13

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Agent A = {ClientAgent@Mobile Device</td>
<td>PresentationPerformer@CRUMPET Access Node FIPA-OS } requests for currently available access methods using an ACL message with QoS attribute as parameter in the content.</td>
</tr>
<tr>
<td>2. MA sends back an agree-ACL message.</td>
<td>Agent A receives the agree-ACL message.</td>
</tr>
<tr>
<td>3. MA senses the network according to the use case 1.1 and finds out the currently available networks.</td>
<td>MA gains knowledge about the currently available networks.</td>
</tr>
<tr>
<td>4. MA asks QoS from every available network.</td>
<td>A list of current QoS values for every available network is constructed.</td>
</tr>
<tr>
<td>5. MA sorts the list according to the given QoS attribute.</td>
<td>A sorted list of available access methods is constructed.</td>
</tr>
<tr>
<td>6. MA creates an inform-ACL message with access method list as content and sends it back to agent A.</td>
<td>Agent A receives the inform-ACL message.</td>
</tr>
</tbody>
</table>
Use Case: 1.5 Subscribe To QoS Changes

Actors: MA@Mobile Device, MA@CRUMPET Access Node FIPA-OS, ClientAgent@Mobile Device, PresentationPerformer@CRUMPET Access Node FIPA-OS, CA@Mobile Device, CA@CRUMPET Access Node FIPA-OS.

Purposes: This use case shows how an agent can subscribe to changes in the QoS.

Cross References: R2.1, R2.2, R2.3, R2.4

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Agent A = {ClientAgent@Mobile Device</td>
<td>PresentationPerformer@CRUMPET Access Node FIPA-OS, CA@Mobile Device</td>
</tr>
<tr>
<td>2. MA stores the agent identifier with the subscription (QoS attributes and constraints) in a database.</td>
<td>The subscription of agent A is stored.</td>
</tr>
<tr>
<td>3. MA detects a change in QoS.</td>
<td>-</td>
</tr>
<tr>
<td>4. MA checks if the change is significant enough to be informed to subscribers.</td>
<td>MA finds that the change in QoS satisfies the constraints Agent A has defined in its subscription.</td>
</tr>
<tr>
<td>5. MA informs Agent A about the change.</td>
<td>Agent A gains knowledge about the changed QoS.</td>
</tr>
</tbody>
</table>

Use Case: 1.6 Query QoS

Actors: MA@Mobile Device, MA@CRUMPET Access Node FIPA-OS, ClientAgent@Mobile Device, PresentationPerformer@CRUMPET Access Node FIPA-OS, CA@Mobile Device, CA@CRUMPET Access Node FIPA-OS

Purposes: Agent A can query the MA about the current QoS using QosInformation function.

Cross References: R2.1, R2.5, R2.6, R2.7, R2.8, R2.9, R2.10, R2.11

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Agent A = {ClientAgent@Mobile Device</td>
<td>PresentationPerformer@CRUMPET Access Node FIPA-OS, CA@Mobile Device</td>
</tr>
<tr>
<td>2. MA sends an agree-ACL message back to Agent A.</td>
<td>Agent A receives the agree-ACL message.</td>
</tr>
<tr>
<td>3. MA measures/calculates the QoS according to the use case 1.3.</td>
<td>QoS-values are measured/calculated.</td>
</tr>
<tr>
<td>4. MA creates an inform-ACL message with QoS-values in content.</td>
<td>Inform-ACL message is created.</td>
</tr>
<tr>
<td>5. MA sends the reply back to the Agent A.</td>
<td>Agent A receives the reply.</td>
</tr>
</tbody>
</table>
4.3.2 Controlling

Use Case: 2.1 Open Comm Channel

Actors: ClientAgent@Mobile Device, CA@Mobile Device, CA@CRUMPET Access Node FIPA-OS

Purposes: This use case shows how CA opens a communication channel.

Cross References: R3.3, R3.7

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ClientAgent@Mobile Device asks CA to open a wireless communication channel¹.</td>
<td>CA receives a request to open a communication channel.</td>
</tr>
<tr>
<td>2. CA verifies that the wireless access is available by asking MA.</td>
<td>MA informs that the channel is available.</td>
</tr>
<tr>
<td>3. CA accesses the user profile according to the use case 6.1 to find out the parameters for the wireless connection.</td>
<td>The parameters are received from the user profile.</td>
</tr>
<tr>
<td>4. CA brings up the wireless connection according to the parameters.</td>
<td>Wireless connection is brought up.</td>
</tr>
<tr>
<td>5. ClientAgent@Mobile Device is informed that the channel is open.</td>
<td>ClientAgent@Mobile Device begins to use the wireless channel.</td>
</tr>
</tbody>
</table>

¹ Here the wireless communication channel is GSM, HSCSD, WLAN, GPRS or UMTS.
Use Case: 2.2 Close Comm Channel

Actors: ClientAgent@Mobile Device, CA@Mobile Device, CA@CRUMPET Access Node FIPA-OS

Purposes: This use case shows how CA closes a communication channel.

Cross References: R3.4, R3.7

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Agent A = ClientAgent@Mobile Device asks CA to close a wireless communication channel1.</td>
<td>-</td>
</tr>
<tr>
<td>2. CA makes the decision about closing the channel.</td>
<td>Decision about closing the channel is made, either yes or no.</td>
</tr>
<tr>
<td>3. Based on the decision, the wireless channel is either closed or not closed.</td>
<td>-</td>
</tr>
<tr>
<td>4. Agent A is informed about the decision.</td>
<td>-</td>
</tr>
</tbody>
</table>

1 Here the wireless communication channel is GSM, HSCSD, WLAN, GPRS or UMTS.

Use Case: 2.3 Activate Transport

Actors: ClientAgent@Mobile Device, CA@Mobile Device

Purposes: This use case shows how CA activates a transport.

Cross References: R3.1, R3.7

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ClientAgent@Mobile Device asks CA@Mobile Device to activate a transport, e.g WAP MTP.</td>
<td>-</td>
</tr>
<tr>
<td>2. CA@Mobile Device brings up the transport.</td>
<td>-</td>
</tr>
<tr>
<td>3. ClientAgent@Mobile Device is informed that the transport is active.</td>
<td>ClientAgent@Mobile Device begins to use transport.</td>
</tr>
</tbody>
</table>

Use Case: 2.4 Deactivate Transport

Actors: ClientAgent@Mobile Device, CA@Mobile Device, CA@CRUMPET Access Node FIPA-OS

Purposes: This use case shows how CA deactivates a transport.

Cross References: R3.2, R3.7

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ClientAgent@Mobile Device asks CA to deactivate a transport, e.g WAP MTP.</td>
<td>-</td>
</tr>
<tr>
<td>2. CA checks if other agents are using the transport.</td>
<td>Decision about deactivating the transport is made, either yes or no.</td>
</tr>
<tr>
<td>3. Based on the decision, the transport is either deactivated or left active.</td>
<td>-</td>
</tr>
<tr>
<td>4. Agent A is informed about the decision.</td>
<td>-</td>
</tr>
</tbody>
</table>
### 4.3.3 WAP messaging

#### 3.1 WAP Message From Mobile Device To CRUMPET Access Node

**Figure 5: Use cases in WAP messaging package.**

<table>
<thead>
<tr>
<th>Use Case:</th>
<th>3.1 WAP Message From Mobile Device to CRUMPET Access Node</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors:</strong></td>
<td>ACC@Mobile Device², ACC@CRUMPET Access Node FIPA-OS</td>
</tr>
<tr>
<td><strong>Purposes:</strong></td>
<td>This use case outlines, how Mobile Device uses WAP transport in order to send an ACL message to the CRUMPET Access Node.</td>
</tr>
<tr>
<td><strong>Cross References:</strong></td>
<td>R4.1, R4.4, R5.*, R6.1, R6.5</td>
</tr>
</tbody>
</table>

**Typical course of events:**

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ACC@Mobile Device receives an ACL message, which is to be sent using WAP transport.</td>
<td>-</td>
</tr>
<tr>
<td>2. ACC@Mobile Device creates a WSP Post PDU.</td>
<td>WSP Post PDU is created.</td>
</tr>
<tr>
<td>3. ACC@Mobile Device encodes the ACL message to bit-efficient ACL message.</td>
<td>Bit-efficient ACL message is created.</td>
</tr>
<tr>
<td>4. ACC@Mobile Device constructs a bit-efficient envelope out of the envelope of the ACL message. If no envelope is given, one is constructed based on the ACL message.</td>
<td>Bit-efficient envelope is constructed.</td>
</tr>
<tr>
<td>5. ACC@Mobile Device sets the URI as the address of ACC@CRUMPET Access Node FIPA-OS.</td>
<td>URI is set.</td>
</tr>
<tr>
<td>6. ACC@Mobile Device sets the content type header as &quot;multipart/mixed&quot;.</td>
<td>Content type header is set.</td>
</tr>
<tr>
<td>7. ACC@Mobile Device sets the content length header as the number of bytes in the content of the WSP Post PDU.</td>
<td>Content length header is set.</td>
</tr>
<tr>
<td>8. ACC@Mobile Device sets the bit-efficient header and ACL message as the payload to the WSP Post PDU.</td>
<td>WSP Post PDU's payload is set to bit-efficient envelope and ACL message.</td>
</tr>
<tr>
<td>9. ACC@Mobile Device sends the WSP Post PDU using client-side WAP stack.</td>
<td>WAP Gateway receives the PDU.</td>
</tr>
<tr>
<td>10. WAP Gateway forwards the PDU as HTTP request to the receiving HTTPReceiver residing on the CRUMPET Access Node.</td>
<td>HTTPReceiver receives the request.</td>
</tr>
<tr>
<td>11. HTTPReceiver sends back an empty WSP Reply.</td>
<td>WAP Gateway forwards the WSP reply to the Mobile Device's client side WAP stack.</td>
</tr>
</tbody>
</table>

"WTP layer of client side WAP stack takes care of sending an acknowledgement back to the WAP Gateway in order to guarantee the reliable message transfer."

² The ACC refers to a “logical” ACC, which means ACC-like functionality implemented in mobile device. The “real” ACC of the MicroFIPA-OS resides on the CRUMPET Access Node FIPA-OS. The distinction here is made because every FIPA-OS agent has its own MTS thereby making it impossible to address the entity in the MicroFIPA-OS, which handles the communication with the CRUMPET Access Node. If it is decided that the MicroFIPA-OS shares MTS among agents, the ACC here can be referred as MTS.
12. HTTPReceiver extracts the bit-efficient ACL message and the bit-efficient Envelope and forwards them to the ACC@CRUMPET Access Node FIPA-OS.

13. ACC@CRUMPET Access Node FIPA-OS decodes the bit-efficient envelope and ACL message.

<table>
<thead>
<tr>
<th>Use Case:</th>
<th>3.2 WAP Message From CRUMPET Access Node To Mobile Device</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors:</strong></td>
<td>ACC@Mobile Device(^3), ACC@CRUMPET Access Node FIPA-OS</td>
</tr>
<tr>
<td><strong>Purposes:</strong></td>
<td>This use case outlines, how gateway uses WAP transport in order to send an ACL message to the Mobile Device.</td>
</tr>
<tr>
<td><strong>Cross References:</strong></td>
<td>R4.1, R4.4, R5.*, R6.1, R6.5</td>
</tr>
</tbody>
</table>

**Typical course of events:**

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ACC@CRUMPET Access Node FIPA-OS receives an ACL message, which is to be sent using WAP transport.</td>
<td>Bit-efficient ACL message is created.</td>
</tr>
<tr>
<td>2. ACC@CRUMPET Access Node FIPA-OS encodes the ACL message to bit-efficient ACL message.</td>
<td>Bit-efficient envelope is constructed.</td>
</tr>
<tr>
<td>3. ACC@CRUMPET Access Node FIPA-OS constructs a bit-efficient envelope out of the envelope of the ACL message. If no envelope is given, one is constructed based on the ACL message.</td>
<td>WAP Gateway receives the request.</td>
</tr>
<tr>
<td>4. ACC@CRUMPET Access Node FIPA-OS constructs a PAP request and sends it as HTTP to the WAP Gateway.</td>
<td>Mobile Device's client side WAP stack receives the request.</td>
</tr>
<tr>
<td>5. WAP Gateway forwards the request to the Mobile Device's client side WAP stack.</td>
<td>WTP layer of Mobile Device's client side WAP stack takes care of sending an acknowledgement to the WAP Gateway in order to guarantee the reliable message transfer.</td>
</tr>
<tr>
<td>6. ACC@Mobile Device gets the WSP Push PDU's payload and extracts the bit-efficient envelope and ACL message.</td>
<td>Bit-efficient envelope and ACL message are extracted.</td>
</tr>
<tr>
<td>7. ACC@Mobile Device decodes the bit-efficient envelope and ACL message.</td>
<td>Bit-efficient envelope and ACL message are decoded and routed to the receiver.</td>
</tr>
</tbody>
</table>

\(^3\) The ACC refers to a “logical” ACC, which means ACC-like functionality implemented in mobile device. The “real” ACC of the MicroFIPA-OS resides on the CRUMPET Access Node FIPA-OS. The distinction here is made because every FIPA-OS agent has its own MTS thereby making it impossible to address the entity in the MicroFIPA-OS, which handles the communication with the CRUMPET Access Node. If it is decided that the MicroFIPA-OS shares MTS among agents, the ACC here can be referred as MTS.
4.3.4 Disconnection and buffering

Figure 6: Use cases in disconnection and buffering package.

<table>
<thead>
<tr>
<th>Use Case:</th>
<th>4.1 Disconnection and reconnection.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors:</td>
<td>MA@Mobile Device, MA@CRUMPET Access Node FIPA-OS, CA@Mobile Device, CA@CRUMPET Access Node FIPA-OS</td>
</tr>
<tr>
<td>Purposes:</td>
<td>This use case show in the case of disconnection and reconnection.</td>
</tr>
<tr>
<td>Cross References:</td>
<td>R6.6, R6.7, R6.8</td>
</tr>
</tbody>
</table>

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Wireless link is for some reason disconnected.</td>
<td>MA@Mobile Device and MA@CRUMPET Access Node detect the disconnection.</td>
</tr>
<tr>
<td>2. MA@Mobile Device and MA@CRUMPET Access Node check all the subscribed agents interested in notification about disconnection.</td>
<td>MA@Mobile Device finds both CA@Mobile Device and ACC@Mobile Device whereas MA@CRUMPET Access Node finds both CA@CRUMPET Access Node and ACC@CRUMPET Access Node.</td>
</tr>
<tr>
<td>3. CA@Mobile Device, ACC@Mobile Device, CA@CRUMPET Access Node and ACC@CRUMPET Access Node are informed about the disconnection.</td>
<td>CA@Mobile Device and CA@CRUMPET Access Node may try to establish the same MTC again or try some other MTC.</td>
</tr>
<tr>
<td>4. ACC@Mobile Device and ACC@CRUMPET Access Node begins to buffer message according to use case 4.2.</td>
<td>Messages are buffered while disconnected.</td>
</tr>
<tr>
<td>5. In this use case CA@Mobile Device re-establishes the same MTC again.</td>
<td>MA@Mobile Device and MA@CRUMPET Access Node detect that the MTC is up.</td>
</tr>
<tr>
<td>6. Buffered messages are sent to the peer.</td>
<td>Buffers are emptied.</td>
</tr>
</tbody>
</table>
Use Case: 4.2 Message buffering

**Actors:** ACC@Mobile Device\(^4\), ACC@CRUMPET Access Node FIPA-OS

**Purposes:** This use case shows, how the message buffering happens while disconnected.

**Cross References:** R6.7

Typical course of events:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>CONSEQUENTIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ACC@Mobile Device and ACC@CRUMPET Access Node are informed about disconnection.</td>
<td>ACC@Mobile Device and ACC@CRUMPET Access Node begin to buffer messages.</td>
</tr>
<tr>
<td>2. All outgoing messages at both ACC@Mobile Device and ACC@CRUMPET Access Node are stored to the buffer and not given to the MTP for sending.</td>
<td>Messages stay in the buffer.</td>
</tr>
</tbody>
</table>

\(^4\) The ACC refers to a “logical” ACC, which means ACC-like functionality implemented in the Mobile Device. The “real” ACC of the MicroFIPA-OS resides on the CRUMPET Access Node FIPA-OS. The distinction here is made because every FIPA-OS agent has its own MTS thereby making it impossible to address the entity in the MicroFIPA-OS, which handles the communication with the CRUMPET Access Node. If it is decided that the MicroFIPA-OS shares MTS among agents, the ACC here can be referred as MTS.
5 ARCHITECTURAL OVERVIEW

This section defines the Nomadic Architecture (NA) in CRUMPET. It is divided into subsections, which describe the architecture from different aspects. In section 5.1 a general view to the NA is given. Section 5.2 introduces the CRUMPET Access Node. Section 5.3 describes the mapping of the general view to a logical model. In section 5.4 the Mobile device – CRUMPET Access Node architecture is described in more detail.

5.1 General nomadic architecture

The CRUMPET NA follows the client-mediator-server paradigm. Mobile Devices running MicroFIPA-OS are the clients\(^5\), which are wirelessly connected to the services residing on the fixed network via a CRUMPET Access Node. The CRUMPET Access Node is an entity in the fixed network. The CRUMPET Access Node hosts a FIPA-OS agent platform being able to mediate agent interactions between the wireless and wired worlds. Mobile Devices are connected to the CRUMPET Access Node and in turn the CRUMPET Access Node FIPA-OS, through a wireless link via wireless service providers. As the wireless service providers may be country-specific, they are kept separate from the CRUMPET Access Node FIPA-OS. The CRUMPET Access Node FIPA-OS connects to the services in the fixed network.

![Diagram of General CRUMPET communication architecture.](image)

In Figure 7 the general communication architecture is outlined. Every trial site providing CRUMPET services has its own CRUMPET Access Node. The CRUMPET Access Node can have one or more wireless access points, for instance WLAN and GPRS, provided by wireless Internet service providers, to which the CRUMPET Access Node FIPA-OS has interfaces.

\(^{5}\) It should be noted that although the Mobile Device is considered as the client in the CRUMPET system, the Mobile Device itself might provide services.
5.2 CRUMPET Access Node

The CRUMPET Access Node is a logical entity providing the Mobile Devices with wireless connection. Once connected, the agents on the Mobile Device are able to access the CRUMPET services residing in the fixed network via the CRUMPET Access Node. For the wireless access, the CRUMPET Access Node has a number of wireless MTPs, which are connected either directly or via a gateway to the wireless service access points. The CRUMPET Access Node hosts the FIPA-OS, which in turn includes the wireless and wired MTPs. Monitor and Control Agents are also present for monitoring and controlling the wireless link, respectively. The CRUMPET Access Node is depicted in Figure 8.

![Figure 8: The CRUMPET Access Node.](image)

The CRUMPET Access Node is realised so that the wireless MTP will be WAP MTP, which will use WAP Gateway as the wireless gateway. At least in the first phase, the WAP Gateway will reside on the CRUMPET Access Node and the wireless access point accepts GSM, HSCSD and GPRS connections. The wired MTP can be for instance IIOP MTP.
5.3 CRUMPET agent architecture

Deliverable 1.1 [CRU01b] defines a picture of the general architecture of CRUMPET. As the Deliverable 1.1 is not a design document, the architecture defined in it does not state how the architecture could be designed and implemented. As the goal of the CRUMPET project is to evaluate the use of agent technology in the design and implementation of tourism-related services, the design has to be based on agents and agent interactions. In Figure 9 the agent architecture of CRUMPET is depicted. The entities in the architecture of D1.1 are replaced with agents, and the interactions (agent communication using ACL) are presented as arrows indicating the direction of data flow. Solid arrows indicate ACL messages and dashed arrows imply proprietary communication. In the figure, agents are shown in a dark colour, whereas non-agent entities are in a lighter colour.

![Figure 9: CRUMPET agent architecture.](image)

This document specifies only the design of the NA. Therefore, the back-end part in Figure 9 is out of the scope of this document. The architecture of this part will be designed in Deliverables 1.4, 1.5, 1.8 and 1.9. Also, the design for the "GPS Wrapper" in the Mobile Device is out of the scope of this document, but will be designed in D1.4. In the following subsection a deeper look at the Mobile Device - CRUMPET Access Node architecture is taken.
5.4 Mobile Device - CRUMPET Access Node architecture

In Figure 10 the Mobile Device - CRUMPET Access Node architecture is presented. In the figure, the components WP3 needs to design and implement are shown in the darkest colour.

The Mobile Device runs a MicroFIPA-OS [CRU00b] agent platform, which includes a Micro-MTS enabling WAP communication between the Mobile Device and the CRUMPET Access Node. The WAP transport in the Mobile Device has an interface to a client side WAP stack, which includes three layers: WSP, WTP and WDP. At the server side, there is a fully-fledged WAP gateway, which is connected to the MTS of the CRUMPET Access Node FIPA-OS. The WAP gateway is separated from the CRUMPET Access Node FIPA-OS allowing a variety of available WAP gateways to be used\(^6\). Also, as it may be inconvenient or even impossible to have CRUMPET software running on the Internet operator’s and its WAP gateway’s domain, we chose this approach, in which only HTTP-connection from the WAP gateway to the CRUMPET Access Node is needed. However, this approach does not restrict CRUMPET from using its own WAP gateway residing on the same domain as the CRUMPET Access Node, if even the HTTP-connection from Internet operator's domain is impossible for instance for security reasons or because the operator's WAP gateway is not suitable in terms of functionality. The design for WAP transport that is used in CRUMPET is specified in section 9.

On both the Mobile Device and the CRUMPET Access Node there is a Monitor Agent (MA), which monitors the MTC and MTP. It provides an ACL interface for other agents to access its services. The design of the MA is specified in section 6.

\(^6\) It must be noted though that the WAP gateway must support PUSH-functionality.
Because Monitor Agents need information about the QoS over the wireless link, the Congestion Manager (CM) [And00, Bal99] is used to provide such information. The CM is implemented in the LINUX kernel and it replaces the LINUX TCP flow control algorithms. Also, both TCP (TCP/CM) and UDP (UDP/CM) are implemented as in-kernel clients for the CM, thus allowing all the communication be managed by the CM. Monitor Agents access the services provided by the CM by using the libcm library, which in turn accesses the CM using CM-specific control sockets. For a more detailed introduction to the CM, see Appendix D. The design for using a CM in CRUMPET is specified in section 8.

Control Agents on both the Mobile Device and the CRUMPET Access Node (CAN) take care of controlling the wireless link. Both of them provide an ACL interface for other agents to access the services they provide. The CA residing on the Mobile Device includes also a Connector and a DeviceHandler. The CA uses the Connector to make a PPP- or WLAN connection to the CAN. The SerialHandler takes care of monitoring PCMCIA slots and serial ports and their status so that the CA is informed whenever (for example) a PCMCIA card is inserted or removed. The design of the CA is specified in section 7.

In the case of disconnection, message buffering takes place on both the CRUMPET Access Node FIPA-OS and the MicroFIPA-OS. The Mobile Device's message buffer for incoming messages resides on the CRUMPET Access Node FIPA-OS and for outgoing messages on the Mobile Device itself. For the detailed design of message buffering, see section 12 on page 120.

In Figure 10 the Client Agent, A/V Player and HTML Browser are also depicted. The Client Agent is basically a CRUMPET agent, which provides the graphical user interface to the user and accesses and uses the services provided by the Monitor Agent, Control Agent or MicroFIPA-OS and its Micro-MTS (WAP transport). The A/V Player and HTML Browser are applications that are not CRUMPET agents but provide output to the user and access the network.
6 NOMADIC APPLICATION SUPPORT ONTOLOGY

6.1 FIPA-NAS ontology in CRUMPET

FIPA Nomadic Application Support (NAS) ontology is used for representing the QoS of the MTS in the context of nomadic application support [FIPA00a]. The FIPA-NAS ontology divides into frames and functions/predicates. Frames represent the classes of objects in the domain of discourse within the framework of the FIPA-NAS ontology.

In CRUMPET, the FIPA-NAS ontology is implemented, although some of the parameters of the QoS are left out. Table 9 defines the conformance clause for the implementation of frames of NAS ontology in CRUMPET and Table 10 defines the predicates and functions (for more detailed description of the frames, predicates and functions, see [FIPA00a]).

<table>
<thead>
<tr>
<th>Frame</th>
<th>Parameter</th>
<th>Supported</th>
<th>Reason for not supporting</th>
</tr>
</thead>
<tbody>
<tr>
<td>qos</td>
<td>line-rate</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>throughput</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>throughput-std-dev</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rtt</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rtt-std-dev</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delay</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delay-std-dev</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mean-up-time</td>
<td></td>
<td>Not considered relevant for neither users nor CRUMPET agents.</td>
</tr>
<tr>
<td></td>
<td>omission-rate</td>
<td></td>
<td>Not considered relevant for neither users nor CRUMPET agents.</td>
</tr>
<tr>
<td></td>
<td>ber</td>
<td></td>
<td>Not considered relevant for neither users nor CRUMPET agents.</td>
</tr>
<tr>
<td></td>
<td>frame-error-rate</td>
<td></td>
<td>Not considered relevant for neither users nor CRUMPET agents.</td>
</tr>
<tr>
<td></td>
<td>conn-setup-delay</td>
<td></td>
<td>Not considered relevant for neither users nor CRUMPET agents.</td>
</tr>
<tr>
<td></td>
<td>conn-setup-failure-prob</td>
<td></td>
<td>Not considered relevant for neither users nor CRUMPET agents.</td>
</tr>
<tr>
<td></td>
<td>available</td>
<td>√</td>
<td>(This is CRUMPET-specified addition to NAS ontology.)</td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>rate-value</td>
<td>direction</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unit</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>time-value</td>
<td>direction</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unit</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>probability-value</td>
<td>direction</td>
<td></td>
<td>Not supported because none of the supported qos-parameters needs this.</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td></td>
<td>Not supported because none of the supported qos-parameters needs this.</td>
</tr>
<tr>
<td>change-constraint</td>
<td>value</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>time-constraint</td>
<td>type</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>comm-channel</td>
<td>name</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target-addr</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>options</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>transport-protocol</td>
<td>name</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td></td>
<td>gw-addr</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dest-addr</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td>options</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>transports</td>
<td>send  Not supported, because use-function is not supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recv  Not supported, because use-function is not supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>msg-representation</td>
<td>name  Not supported, because use-function is not supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>options Not supported, because use-function is not supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>msg-rep-selection</td>
<td>send  Not supported, because use-function is not supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recv  Not supported, because use-function is not supported.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Predicate/function</th>
<th>Supported by</th>
<th>Supported</th>
<th>Reason for not supporting</th>
</tr>
</thead>
<tbody>
<tr>
<td>qos-information</td>
<td>MA</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>qos-notification</td>
<td>MA</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>open-comm-channel</td>
<td>CA</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>close-comm-channel</td>
<td>CA</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>activate</td>
<td>CA</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>deactivate</td>
<td>CA</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>use</td>
<td>CA</td>
<td>√</td>
<td>In CRUMPET there is only one MTP over the wireless link. Thus negotiations for the MTP to be used are irrelevant.</td>
</tr>
</tbody>
</table>

Exceptions are defined in [FIPA00a] and are supported in CRUMPET.

6.2 Interfaces to other agents

NAS ontology is used when interacting with the MA and CA. Figure 11 depicts the interfaces between the NAS ontology and other agents. The NAS ontology package contains all the NAS ontology-related classes. The NASOntology package is specified in more detail in section 6.4. The FIPA-OS Agent package represents any FIPA-OS agent. Designs for Monitoring and Controlling are specified in upcoming sections.

![Figure 11: NAS ontology interfaces.](image-url)
6.3 Design principles

All NasOntology classes are created by the NasOntologyFactory according to the Factory-pattern [Gra98]. Therefore, the users of NasOntologyFactory do not have to know the details of how the concrete NasOntology class is created. All they do is pass either ACL message or Content object to NasOntologyFactory, which in turn takes care of creating and returning the NasOntology object. There are six concrete NasOntology objects in NAS: QosInformationOntology, QosNotificationOntology, OpenCommChannelOntology, CloseCommChannelOntology, ActivateOntology and DeactivateOntology. The reason for having specialised ontology objects is that they can be used according to the command pattern [Gra98] as command objects when executing NAS-related functions.

NasOntologyFactory delegates the NasOntology object creation to the specialised NasOntology objects which all implement the NasOntology interface. The specialised concrete NasOntology delegates the parsing of the corresponding ontology to a specialised NasOntologyParser object, which is able, using the tokens provided via the NasParser interface, to construct the concrete NasOntologyObject.

The design rationale for separating the lexical and grammatical parsers is that by having the lexical tokeniser separated from the grammar, we are not dependent on the format of the file to be parsed. The specialised lexical tokenisers are just responsible to provide NAS tokens from the file format they are specialised to tokenising.

The specialised grammar parsers on the other hand could be combined into a one large NAS ontology grammar parser, but it is easier to implement a more specialised parser than to put all the parsing functionality for every NAS ontology object into one parser. As NAS ontologies share frames to some extent, all the common parsing functionality will be collected in only one place.

6.4 Class diagrams

6.4.1 QosInformationOntology

The class diagram for QosInformationOntology is presented in Figure 12. For detailed definitions for the classes, see section 6.5 on page 43.

![Figure 12: QosInformationOntology class diagram.](image-url)
6.4.2 QosNotificationOntology
The class diagram for QosNotificationOntology is presented in Figure 13. For detailed definitions for
the classes, see section 6.5 on page 43.

Figure 13: QosNotificationOntology class diagram.

6.4.3 OpenCommChannelOntology
The class diagram for OpenCommChannelOntology is presented in Figure 14. For detailed definitions
for the classes, see section 6.5 on page 43.

Figure 14: OpenCommChannelOntology class diagram.
6.4.4 CloseCommChannelOntology

The class diagram for CloseCommChannelOntology is presented in Figure 15. For detailed definitions for the classes, see section 6.5 on page 43.

![Figure 15: CloseCommChannelOntology class diagram.](image)

6.4.5 ActivateOntology

The class diagram for ActivateOntology is presented in Figure 16. For detailed definitions for the classes, see section 6.5 on page 43.

![Figure 16: ActivateOntology class diagram.](image)
6.4.6 DeactivateOntology

The class diagram for DeactivateOntology is presented in Figure 17. For detailed definitions for the classes, see section 6.5 on page 43.

![DeactivateOntology class diagram](image)

Figure 17: DeactivateOntology class diagram.

6.5 Class definitions

6.5.1 NasOntologyFactory

*Package*: crumpet.nas.ontology

*Syntax*: static class NasOntologyFactory

*Direct Known Subclasses*: none

*All Implemented Interfaces*: none

The NASOntologyFactory class is able to create an NAS ontology object out of an FIPA-OS ACL object. The ontology object to be returned is either a valid NAS ontology object or null object, if the NAS ontology object could not be created for one reason or another.

*Methods*:

```java
public static NasOntology create (ACL acl)
```

This method creates a NasOntology object out of the content field of the ACL message.

*Parameters*: acl —The ACL message from which the content field is taken and the NasOntology object is created.

*Returns*: NasOntology object or null, if object could not be created.
6.5.2 NasOntology

Package: crumpet.nas.ontology
Syntax: public interface NasOntology
Direct Known Subclasses: none.
All Implemented Interfaces: none.

NasOntology is an interface defining three mandatory methods, which all the implementing classes must implement.

Methods:

public NasOntology create (ACL acl)
This method creates a NasOntology object out of the content field of the ACL message.

Parameters: acl —The ACL message from which the content field is taken and the NasOntology object is created.

Returns: NasOntology object or null, if object could not be created.

public String toString ()
This method prints out a String representation of the ontology.

Returns: Ontology as String.

6.5.3 NasParser

Package: crumpet.nas.ontology
Syntax: public interface NasParser
Direct Known Subclasses: none.
All Implemented Interfaces: none.

NasParser defines a common interface to all the parsers used to parse NAS ontologies. By having an NasParser interface, the users of the parser do not have to know the format of the ontology and also, NAS ontology classes are not dependent on any format of the ontology representation. A possible parser for NAS ontologies could be SL2 Parser [FIPA00f]. At the time of writing this Deliverable there is no agreement among the CRUMPET partners on which content language will be used.

Methods:

public String getToken ()
This method gets the next token from the ontology representation.

Returns: Next token as String.

6.5.4 SL2Parser

Package: crumpet.nas.ontology
Syntax: public class SL2Parser implements NasParser
Direct Known Subclasses: none.
All Implemented Interfaces: NasParser

The SL2Parser parses the SL2 representation of the NAS ontology and is able to provide tokens from it. It must be noted that SL2 is not the content language CRUMPET is committed to use.
Constructors:

public SL2Parser (OutputStream out)

Parameters: out — OutputStream, from which the tokens are read and parsed.

public SL2Parser ()

Methods:

public String getToken ()

This method gets the next token from the SL2 representation.

Returns: Next token as String.

public void setStream (OutputStream out)

Sets the OutputStream, from which the tokens are read and parsed.

Parameters: out — OutputStream, from which the tokens are read and parsed.

6.5.5 CommChannel

Package: crumpet.nas.ontology
Syntax: public class CommChannel
Direct Known Subclasses: none.
All Implemented Interfaces: none.

CommChannel represents the CommChannel frame as specified in [FIPA00a],

Constructors:

public CommChannel ()

Attributes:

String name
String targetAddr
String options

6.5.6 TransportProtocol

Package: crumpet.nas.ontology
Syntax: public class TransportProtocol
Direct Known Subclasses: none.
All Implemented Interfaces: none.

TransportProtocol represents the TransportProtocol frame as specified in [FIPA00a],

Constructors:

public TransportProtocol ()

Attributes:

String name
String gwAddr
String destAddr
String options
6.5.7 NasQos
For definition, see section 7.5.13 on page 66.

6.5.8 TimeValue
Package: crumpet.nas.ontology
Syntax: public class TimeValue
Direct Known Subclasses: none.
All Implemented Interfaces: none.

TimeValue represents the TimeValue frame as specified in [FIPA00a].

Constructors:
public TimeValue ()

Attributes:
String direction
String unit
String value

6.5.9 RateValue
Package: crumpet.nas.ontology
Syntax: public class RateValue
Direct Known Subclasses: none.
All Implemented Interfaces: none.

RateValue represents the RateValue frame as specified in [FIPA00a].

Constructors:
public RateValue ()

Attributes:
String direction
String unit
String value

6.5.10 ChangeConstraint
Package: crumpet.nas.ontology
Syntax: public class ChangeConstraint
Direct Known Subclasses: none.
All Implemented Interfaces: none.

ChangeConstraint represents the ChangeConstraint frame as specified in [FIPA00a].

Constructors:
public ChangeConstraint ()

Attributes:
String value
6.5.11 TimeConstraint

Package: crumpet.nas.ontology
Syntax: public class TimeConstraint
Direct Known Subclasses: none.
All Implemented Interfaces: none.

TimeConstraint represents the TimeConstraint frame as specified in [FIPA00a].

Constructors:
   public TimeConstraint ()

Attributes:
   String type
   String value

6.5.12 QosInformationOntology

Package: crumpet.nas.ontology
Syntax: public class QosInformationOntology implements NasOntology
Direct Known Subclasses: none.
All Implemented Interfaces: NasOntology.

QosInformationOntology encapsulates the qos-information domain.

Constructors:
   public QosInformationOntology ()

Methods:
   public NasOntology create (ACL acl)
       This method creates a QosInformationOntology object out of the content field of the ACL message.

       Parameters: acl — The ACL message from which the content field is taken and the QosInformationOntology object is created.

       Returns: NasOntology—typed QosInformationOntology object or null, if object could not be created.

   public String toString ()
       This method prints out a String representation of itself.

       Returns: QosInformationOntology as String.
6.5.13 QosInformationParser

Package: crumpet.nas.ontology
Syntax: public class QosInformationParser
Direct Known Subclasses: none.
All Implemented Interfaces: none.

QosInformationParser is responsible to construct a QosInformationOntology object out of the sequence of tokens it reads through the NasParser interface.

Constructors:
public QosInformationParser ()

Methods:
public parse (NasParser tokeniser)
Parameters: tokeniser — The NasParser object, which provides the tokens.

6.5.14 QosNotificationOntology

Package: crumpet.nas.ontology
Syntax: public class QosNotificationOntology implements NasOntology
Direct Known Subclasses: none.
All Implemented Interfaces: none.

QosNotificationOntology encapsulates the qos-notification domain.

Constructors:
public QosNotificationOntology ()

Methods:
public NasOntology create (ACL acl)
Parameters: acl — The ACL message from which the content field is taken and the QosNotificationOntology object is created.

Returns: NasOntology—typed QosNotificationOntology object or null, if object could not be created.

public String toString ()
This method prints out a String representation of itself.

Returns: QosNotificationOntology as String.
6.5.15 QosNotificationParser

Package: crumpet.nas.ontology
Syntax: public class QosNotificationParser
Direct Known Subclasses: none.
All Implemented Interfaces: none.

QosNotificationParser is responsible to construct a QosNotificationOntology object out of the sequence of tokens it reads through the NasParser interface.

Constructors:
  public QosNotificationParser ()

Methods:
  public parse (NasParser tokeniser)
  Parameters: tokeniser — The NasParser object, which provides the tokens.

6.5.16 OpenCommChannelOntology

Package: crumpet.nas.ontology
Syntax: public class OpenCommChannelOntology implements NasOntology
Direct Known Subclasses: none.
All Implemented Interfaces: NasOntology.

OpenCommChannelOntology encapsulates the open-comm-channel domain.

Constructors:
  public OpenCommChannelOntology ()

Methods:
  public NasOntology create (ACL acl)
  This method creates an OpenCommChannelOntology object out of the content field of the ACL message.
  Parameters: acl — The ACL message from which the content field is taken and the OpenCommChannelOntology object is created.
  Returns: NasOntology—typed OpenCommChannelOntology object or null, if object could not be created.

  public String toString ()
  This method prints out a String representation of itself.
  Returns: OpenCommChannelOntology as String.
6.5.17 OpenCommChannelParser

*Package:* crumpet.nas.ontology

*Syntax:* public class OpenCommChannelParser

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* none.

OpenCommChannelParser is responsible to construct an OpenCommChannelOntology object out of the sequence of tokens it reads through the NasParser interface.

**Constructors:**

public **OpenCommChannelParser** ()

**Methods:**

public **parse** (NasParser tokeniser)

*Parameters:* tokeniser — The NasParser object, which provides the tokens.

6.5.18 CloseCommChannelOntology

*Package:* crumpet.nas.ontology

*Syntax:* public class CloseCommChannelOntology implements NasOntology

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* NasOntology.

CloseCommChannelOntology encapsulates the close-comm-channel domain.

**Constructors:**

public **CloseCommChannelOntology** ()

**Methods:**

public NasOntology **create** (ACL acl)

*This method creates a CloseCommChannelOntology object out of the content field of the ACL message.*

*Parameters:* acl — The ACL message from which the content field is taken and the CloseCommChannelOntology object is created.

*Returns:* NasOntology-typed CloseCommChannelOntology object or null, if object could not be created.

public String **toString** ()

*This method prints out a String representation of itself.*

*Returns:* CloseCommChannelOntology as String.
6.5.19 CloseCommChannelParser

Package: crumpet.nas.ontology
Syntax: public class CloseCommChannelParser
Direct Known Subclasses: none.
All Implemented Interfaces: none.

CloseCommChannelParser is responsible to construct a CloseCommChannelOntology object out of the sequence of tokens it reads through the NasParser interface.

Constructors:
public CloseCommChannelParser ()

Methods:
public parse (NasParser tokeniser)
Parameters: tokeniser — The NasParser object, which provides the tokens.

6.5.20 ActivateOntology

Package: crumpet.nas.ontology
Syntax: public class ActivateOntology implements NasOntology
Direct Known Subclasses: none.
All Implemented Interfaces: NasOntology.

ActivateOntology encapsulates the activate domain.

Constructors:
public ActivateOntology ()

Methods:
public NasOntology create (ACL acl)
Parameters: acl — The ACL message from which the content field is taken and the ActivateOntology object is created.
Returns: NasOntology-typed ActivateOntology object or null, if object could not be created.

public String toString ()
This method prints out a String representation of itself.
Returns: ActivateOntology as String.
6.5.21 ActivateParser

*Package:* crumpet.nas.ontology

*Syntax:* public class ActivateParser

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* none.

ActivateParser is responsible to construct an ActivateOntology object out of the sequence of tokens it reads through the NasParser interface.

*Constructors:*

  public **ActivateParser** ()

*Methods:*

  public **parse** (NasParser `tokeniser`)

  *Parameters:* `tokeniser` — The NasParser object, which provides the tokens.

6.5.22 DeactivateOntology

*Package:* crumpet.nas.ontology

*Syntax:* public class DeactivateOntology implements NasOntology

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* NasOntology.

DeactivateOntology encapsulates the deactivate domain.

*Constructors:*

  public **DeactivateOntology** ()

*Methods:*

  public NasOntology **create** (ACL `acl`)

  *This method creates a DeactivateOntology object out of the content field of the ACL message.*

  *Parameters:* `acl` — The ACL message from which the content field is taken and the DeactivateOntology object is created.

  *Returns:* NasOntology-typed DeactivateOntology object or null, if object could not be created.

  public String **toString** ()

  *This method prints out a String representation of itself.*

  *Returns:* DeactivateOntology as String.
6.5.23 DeactivateParser

*Package:* crumpet.nas.ontology  
*Syntax:* public class DeactivateParser  
*Direct Known Subclasses:* none.  
*All Implemented Interfaces:* none.

DeactivateParser is responsible to construct a DeactivateOntology object out of the sequence of tokens it reads through the NasParser interface.

**Constructors:**

public DeactivateParser ()

**Methods:**

public parse (NasParser tokeniser)  
**Parameters:** tokeniser — The NasParser object, which provides the tokens.
7 MONITOR AGENT

7.1 Description
The Monitor Agent (MA) is an agent that collects QoS information from the wireless link and provides it to other agents by providing an ACL interface for queries. The MA collects the data autonomously and stores the data for first level analysis. First level analysis for the collected QoS data is done by calculating mean values and standard deviations for the data. The MA resides on both the Mobile Device and the CAN.

The MA has two parts. The FIPA-OS specific part is tightly coupled with FIPA-OS tasks while the platform-independent part is common for any platform.

The MA relies on network specific components for QoS collecting. Therefore, the MA is designed so that it is not dependent on any specific way of collecting QoS, but defines a common interface, which the network specific components must implement. Thus, the actual gathering of QoS data can be (for example) measuring, if the network itself cannot provide any QoS data. In CRUMPET, QoS data is provided by Congestion Manager (CM), which is specified in section 8.

The MA provides two ways for requesting the QoS: query and subscription. In query, an agent asks for the current QoS according to the FIPA-Query [FIPA00g] interaction protocol. The answer is based on the current knowledge of the QoS. In subscription, an agent subscribes to the information about changes occurring in the QoS on the wireless link according to the FIPA-Subscribe [FIPA00h] interaction protocol. In this case, the MA maintains a database about subscriptions, and as the QoS changes, it applies subscription matching for the entries in the database. If a match is found, the corresponding agent is informed. Query is described in more detail in section 6.2.1 and subscription in section 6.2.2. The QoS values provided by the MA in CRUMPET are collected in Table 11.

<table>
<thead>
<tr>
<th>QoS value</th>
<th>Collection method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linerate</td>
<td>Stored in network device profile.</td>
<td>The bandwidth in one direction over the link.</td>
</tr>
<tr>
<td>Throughput</td>
<td>Provided by CM.</td>
<td>The number of user data bits successfully transferred in one direction across the link. Successful transfer means that no user data bits are lost, added or inverted in transfer.</td>
</tr>
<tr>
<td>Throughput-std-dev</td>
<td>Calculated(*) by MA.</td>
<td>The current standard deviation of the throughput within a time unit.</td>
</tr>
<tr>
<td>Rtt</td>
<td>Provided by CM.</td>
<td>The round trip time, which is the time, required for a data segment to be transmitted to a peer entity and a corresponding acknowledgement sent back to the originating entity.</td>
</tr>
<tr>
<td>Rtt-std-dev</td>
<td>Calculated(*) by MA.</td>
<td>The current standard deviation of the round-trip time within a time unit.</td>
</tr>
<tr>
<td>Delay</td>
<td>Provided by CM.</td>
<td>The (nominal) time required for a data segment to be transmitted to a peer entity.</td>
</tr>
<tr>
<td>Delay-std-dev</td>
<td>Calculated(*) by MA.</td>
<td>The current standard deviation of the delay within a time unit.</td>
</tr>
<tr>
<td>Available</td>
<td>Observed by MA.</td>
<td>This value is true, if network is available, false otherwise.</td>
</tr>
<tr>
<td>Status</td>
<td>Observed by MA.</td>
<td>The connectivity status of the link. According to the FIPA Nomadic Application Support specification [FIPA00a], <strong>Connected</strong> means that there (at least) logical connection between communicating entities. <strong>Disconnected</strong> means that there is no connection between communicating entities, and the communicating entities are not establishing a connection at the moment. <strong>Connecting</strong> means that there is no connection between communicating entities, but they are currently establishing a connection between them.</td>
</tr>
</tbody>
</table>
(*) The calculation for standard deviation is done periodically by QoSManager (see class definition in section 7.5.14 on page 67) on every \( n \)’th (where the value for \( n \) is implementation-specific) QoS collection of the corresponding QoS value. The samples for the calculation are the last \( 2^n \) samples, which implies that the \( n \) new samples with \( n \) old samples are taken into the calculation. At the time of MA start-up, the initial samples for the calculations are the last \( 2^n \) persistently stored samples.

7.2 External ACL interfaces

7.2.1 Qos-information
An agent asks for quality of service information from an MA using the FIPA-Query [FIPA00g] interaction protocol (see example\(^7\) in Figure 18). The agent may specify either a communication channel or transport protocol to request quality of service information from. The predicate is true, when the values of the QoS parameters defined in the QoS object are true for the given communication channel or transport protocol (i.e., the QoS of communication channel or transport protocol is what stated in the QoS object). Otherwise the predicate is false.

![Figure 18: Qos-information query.](image)

```
[1] {query-ref
 :sender
  {agent-identifier
   :name AgentA@foo.com}
 :receiver (set
  {agent-identifier
   :name MA@foo.com})
 :ontology FIPA-Nomadic-Application
 :language FIPA-SL2
 :protocol FIPA-Query
 :content
 {iota ?x
  {qos-information
   {comm-channel
    :name GPRS}
   {qos
    :throughput ?x)}}}
```

\(^7\) In this example not all the possible messages are taken into account. For a complete specification of the FIPA-Query interaction protocol, see [FIPA00g]. It must be noted that although this example uses SL2, there is no agreement in CRUMPET on which content language should be used. Furthermore, the string-based encoding used is just for clarity; at runtime messages are encoded bit-efficiently.
7.2.2 Qos-notification

An agent subscribes to notifications about changes to the quality of service from an MA using the FIPA-Subscribe [FIPA00h] interaction protocol (see example 8 in Figure 19). The example does not show how the unsubscription is done, because at the time of writing this Deliverable, the specification work for unsubscription is being done. As soon as FIPA agrees on the semantics of unsubscription, an example of it will be added here.

![Diagram showing Qos-notification subscription.]

---

8 In this example not all the possible messages are taken into account. For a complete specification of the FIPA-Subscribe interaction protocol, see [FIPA00h]. It must be noted that although this example uses SL2, there is no agreement in CRUMPET on which content language should be used. Furthermore, the string-based encoding used is just for clarity; at runtime messages are encoded bit-efficiently.
7.3 Constraints and limitations

The main constraint for the design of the MA is the limited processing power and memory resources of the Mobile Device. Especially the MA in the mobile device has to be designed to be as lightweight as possible. One solution is not to have a full-blown MA extending the FIPAOSAgent class in the Mobile Device, but to design it so that the MA functionality is designed as a collection of FIPAOS tasks, which could be reused by agents in need of the MA's services. As the CRUMPET Access Node does not have such limitations as the Mobile Device, we can have a full-blown MA running there utilising the same FIPA-OS tasks as in the Mobile Device. As no prototypes are yet implemented on the runtime environment in the Mobile Device, we cannot say for sure, whether or not the MA in the Mobile Device should be left as a collection of FIPA-OS tasks. However, we do not rule out this possibility from our design.

The amount of threads may cause performance problems, especially if every task runs in its own thread. Therefore, we have to take into account that the number of tasks and threads may have to be reduced to achieve better performance. The design of the Monitor Agent allows for a reduction of the amount of threads used. As soon as the first prototype of the MicroFIPA-OS can be tested for performance, we will have a better understanding about the effects of threading on the performance.

Another constraint is that because the actual gathering of QoS information is very low-level function, the design does not specify any generic way of implementing the QoS gathering. Instead, the design divides into "two layers". The "upper layer" is a common one defining the interfaces, which the "lower level" implementation-specific components must implement. Whilst the "upper layer" is designed to be compatible with any Java enabled device, the "lower level" depends upon the operating system to be used. In CRUMPET, the operating system to be used is LINUX and the "lower layer" QoS gathering is implemented by the CM. This is specified in more detail in upcoming subsection 7.5.11 on page 66.

7.4 Class diagrams

7.4.1 FIPA-OS specific classes

In Figure 20 the classes implementing the FIPAOS task of the MA are depicted. In the CRUMPET Access Node, all the functionality is encompassed in the MonitorAgent, which is inherited from the FIPAOSAgent thereby making it a FIPAOS agent. The MonitorAgent includes a number of tasks, which are inherited from the FIPAOS task allowing the FIPAOS task manager to manage them. If the limitations of the Mobile Device are too restrictive in processing power and the amount of runtime...
memory, the MonitorAgent will not be implemented in the Mobile Device, but instead the tasks will be allowed to be adopted by other (client) agent directly. The Monitor Agent’s tasks utilise the MAFunctions class in executing the Monitor Agent specific functions. The MAFunction mediates the FIPA-OS specific and platform-independent parts.

The MAFunctions class provides a single entry point for accessing the MA services. Thus, it acts as a facade [Gra98] for the underlying MA functionality. The MA tasks do not have to know the details of how an incoming request in ACL message is actually handled; all they have to know is the interface of the MAFunctions for calling the execution method with the NasOntology object as parameter. In return, upon completing the request, MAFunctions class returns the result NasOntology object to the MA task.

MAFunctions class is a singleton [Gra98] class. Therefore, it contains a static method to get a reference to it thus relieving the MA tasks from the need to know how the MAFunctions class should be instantiated. Singleton pattern also assures that there will not be more than one instance of the MAFunctions.

The classes in Figure 20 are specified in more detail in upcoming section.

---

**Figure 20: FIPA-OS specific Monitor Agent classes.**

7.4.2 Platform-independent classes

In Figure 21 the functionality behind the MAFunctions class is presented. Based on the type of request, the MAFunctions delegates the handling of the request to an appropriate handler class, which in the Monitor Agent case are the handlers for qos-information and qos-notification requests. As qos-information requires knowledge of the current QoS, the QosInformation handler uses the QosCollector information in order to get the QoS. The QoS values are encapsulated in the NasQos container class.

QosInformationFunction and QosNotificationFunction implement the actual functions. Both of them implement the NasFunction interface, therefore defining execute-function, which the MAFunctions object call with the NasOntology object as parameter. Thus, the MAFunctions object only needs to
construct an appropriate function class for executing the request implicitly defined by the NasOntology object. This allows strategy pattern-like [Gra98] behaviour, where the MAFunctions object has a set of different strategies for handling the incoming request. This also promotes scalability and reuse, because the function classes can be added and replaced at runtime.

QosInformationFunction is responsible to get the current QoS. It does so by using the QosCollector interface, which provides a method for asking the QoS. The QosCollector interface decouples the QosInformationFunction from the concrete QosCollector classes. Therefore, the design is not dependent upon any specific QoS collecting method. Instead, we can use for instance SimulatedQosCollector, if real world QoS is not available. Also, this design allows the replacement of the QoS collecting function without any modifications to other classes.

QosNotificationFunction stores the subscriptions provided in the NasOntology object it receives from the execute-method. It stores the subscriptions internally. Furthermore, the subscriptions are not stored to persistent storage so that they are cleared every time the MonitorAgent is destroyed.

QosManager implements the periodic updates for the QoS values, matches the subscriptions and informs the matched subscribers. It also stores the collected QoS values for analysis, such as calculations for standard deviations. It stores the history of QoS values to a persistent database (handled by the QosHistory class), so that when the MonitorAgent is restarted, it already has some QoS values for calculations. For updating the current QoS values QosManager asks the current QoS from the QosCollector interface. The subscriptions interface defines the functionality that the concrete SubscriptionStorage implements. The three methods allow adding, removing and matching subscriptions. SubscriptionStorage contains a collection of SubscriptionItem classes implementing the Subscriptions interface.

The classes in Figure 21 are specified in more detail in upcoming section.

![Figure 21: Platform-independent Monitor Agent classes.](image-url)
7.4.3 MA database

The MA stores the history of QoS values into a persistent storage so that when the MA is restarted, it has some previous values for its calculations. QoS values are MTC/MTP specific. The QoS value history is stored to a local file. An example of this is presented in Figure 22, where the format is RDF [W3C01]. However, the format in which the MA database is stored is not limited to RDF. Instead, the same CRUMPET-wide formats especially in the Mobile Device should be used. At the time of writing this Deliverable the final format is not yet decided.

QoS history is managed by the QosHistory class, which is able to read and write the history from/to a persistent storage and allows the QosManager to feed in new NasQos objects.

```xml
<?xml version="1.0"?>
<rdf:RDF xml:lang="en"
  xmlns="http://foo.com/schemas/devices#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:nas="http://foo.com/schemas/devices#">
  <nas:QoSHistory mtc="gsm" mtp="wap" rdf:about="qoshistory">
    <nas:throughput items="5" unit="bit/s">
      9.5 9.2 8.9 9.1 9.2
    </nas:throughput>
    <nas:throughput-std-dev items="5">0.4 0.5 0.2 0.7 1.2</nas:throughput-std-dev>
    <nas:rtt items="5" unit="ms">
      900 980 970 940 970
    </nas:rtt>
    <nas:rtt-std-dev items="5">102 131 100 126 231</nas:rtt-std-dev>
    <nas:delay items="5" unit="ms">
      402 520 420 410 460
    </nas:delay>
    <nas:delay-std-dev items="5">70 45 78 69 87</nas:delay-std-dev>
  </nas:QoSHistory>

  <nas:QoSHistory mtc="wlan" mtp="iiop" rdf:about="qoshistory">
    <nas:throughput items="5" unit="Mbit/s">
      2.3 4.1 1.2 2.2 3.2
    </nas:throughput>
    <nas:throughput-std-dev items="5">0.1 0.2 0.4 0.7 1.1</nas:throughput-std-dev>
    <nas:rtt items="5" unit="ms">
      90 98 97 94 97
    </nas:rtt>
    <nas:rtt-std-dev items="5">20 23 20 22 23</nas:rtt-std-dev>
    <nas:delay items="5" unit="ms">
      14 10 12 97
    </nas:delay>
    <nas:delay-std-dev items="5">2 3 4 2 5</nas:delay-std-dev>
  </nas:QoSHistory>
</rdf:RDF>
```

Figure 22: QoS value history as RDF.
7.5 Class and module definitions

7.5.1 MonitorAgent

Package: crumpet.nas.ma
Syntax: public class MonitorAgent extends FIPAOSAgent
Direct Known Subclasses: none.
All Implemented Interfaces: none.

This class encapsulates all the MA functionality and takes care of registering with AMS and DF.

 Constructors:
  public MonitorAgent (String platform, String name, String ownership)

7.5.2 MAQueryListenerTask

Package: crumpet.nas.ma
Syntax: public class MAQueryListererTask extends Task
Direct Known Subclasses: none.
All Implemented Interfaces: none.

MAQueryListererTask is FIPAOS task, which listens for incoming requests from other agents and upon receiving one, dispatches the appropriate task for taking care of handling the request.

 Constructors:
  public MAQueryListererTask ()

 Methods:
  public void handleQueryRef (Conversation conv)
  This method handles the incoming query for QoS information.
  Parameters: conv — the conversation to which the query belongs.
  MAQueryListererTask should ask the Conversation object for the latest message in order to get the query for QoS.

  public void handleSubscribe (Conversation conv)
  This method handles the incoming subscription.
  Parameters: conv — the conversation to which the subscription belongs.
  MAQueryListererTask should ask the Conversation object for the latest message in order to get the subscription message.

7.5.3 QosInformationTask

Package: crumpet.nas.ma
Syntax: public class QosInformationTask extends Task
Direct Known Subclasses: none.
All Implemented Interfaces: none.

This class is responsible for handling the qos-information query, but delegates most of the functionality to the MAFunctions class. This design decision promotes portability between different FIPA compliant agent platforms. So basically this class hooks the qos-information functionality in FIPA-OS. First an agree message is sent to the requesting agent. Then the actual qos-information query is delegated to MAFunctions and after it has returned with the result NasOntology, an inform message is sent to the requesting agent.
Constructors:

```java
public QosInformationTask ()
```

Methods:

```java
public void getQos (NasOntology ontology)
```

This method is responsible for getting the current QoS. It does so by delegating the responsibility for MAFunctions class.

**Parameters:** ontology – The ontology object representing the qos-information.

---

7.5.4 QosNotificationTask

**Package:** crumpet.nas.ma

**Syntax:**

```java
public class QosNotificationTask extends Task
```

**Direct Known Subclasses:** none.

**All Implemented Interfaces:** none.

This class is responsible for handling the qos-notification subscriptions, but delegates most of the functionality to the MAFunctions class. This design decision promotes portability between different FIPA compliant agent platforms. So basically this class hooks the qos-notification functionality in FIPA-OS. QosNotification is responsible for taking in subscriptions, storing them into a FIPA-OS database and removing subscriptions from the FIPA-OS database.

Constructors:

```java
public QosNotificationTask ()
```

Methods:

```java
public boolean subscribe (NasOntology ontology)
```

This method takes in a subscription and stores it to a FIPA-OS database.

**Parameters:** ontology – The ontology object representing the subscription.

**Returns:** True, if subscription is successfully stored, false otherwise.

```java
public boolean unsubscribe (NasOntology ontology)
```

This method removes the subscription from the FIPA-OS database.

**Parameters:** ontology — The ontology object representing the unsubscription.

**Returns:** True, if subscription is successfully removed, false otherwise.

---

7.5.5 MAFunctions

**Package:** crumpet.nas.ma

**Syntax:**

```java
public class MAFunctions
```

**Direct Known Subclasses:** none.

**All Implemented Interfaces:** none.

MAFunctions acts as a façade, assigning responsibility to QosInformation or QosNotification classes based on the incoming request. This class is a singleton class implementing one common function for executing either qos-information or qos-notification function. It does so by first creating the appropriate object and then assigning the work for to the object.
Constructors:
    public MAFuncions ()

Methods:
    public void execute (NasOntology ontology)
    This method is called with NasOntology as parameter. Based on the type of the
    NasOntology object, either QosInformation or QosNotification class is created and
    the responsibility of executing the function is passed to it.

    Parameters: ontology — The ontology object representing the function.

7.5.6 NasFunction
Package: crumpet.nas.ma
Syntax: public interface NasFunction
Direct Known Subclasses: none.
All Implemented Interfaces: none.

NasFunction interface defines the executing method, which every MA function class must implement.

Methods:
    public NasOntology execute (NasOntology ontology)
    This method is responsible for executing a MA function and must be implemented by
    every implementing subclass.

    Parameters: ontology — The ontology object representing the function.

    Returns: New ontology object representing the result of the executed function.

7.5.7 QosInformationFunction
Package: crumpet.nas.ma
Syntax: public class QosInformationFunction implements NasFunction
Direct Known Subclasses: none.
All Implemented Interfaces: none.

QosInformationFunction gets the current QoS and returns it to the requesting agent. QosInformationFunction uses QosCollector interface for getting the current QoS. It then constructs a
new QosInformationOntology based on the current QoS and returns it.

Constructors:
    public QosInformationFunction ()

Methods:
    public NasOntology execute (NasOntology ontology)
    This method implements the QoS collecting by using the QosCollector interface.

    Parameters: ontology — The ontology object representing the open-comm-channel
    function

    Returns: New ontology object representing the result of the executed function.
7.5.8 QosNotificationFunction

Package: crumpet.nas.ma
Syntax: public class QosNotificationFunction extends NasFunction
Direct Known Subclasses: none.
All Implemented Interfaces: none.

QosNotificationFunction implements the functionality to take in subscriptions, storing them and removing them.

Constructors:
public QosNotificationFunction ()

Methods:
public void execute ()
This method implements the subscription for QoS changes.

Parameters: ontology — The ontology object representing the qos-notification function.

Returns: New ontology object representing the result of the executed function.

7.5.9 QosCollector

Package: crumpet.nas.ma
Syntax: public interface QosCollector
Direct Known Subclasses: none.
All Implemented Interfaces: none.

QosCollector defines the interface every concrete QoS collecting classes must implement. QosCollector interface is used for decoupling the QoS collecting interface from different kinds of implementations. Design rationale for this is that the users of this interface do not have to know the details of how the actual QoS collecting is applied. Also, this allows changes or even replacement of the actual QoS collecting engine so that for instance simulated QoS can be used if real world QoS data is not available (for instance in the case of UMTS).

Methods:
public NasQos getQos ()
This is the only method all the concrete QoS collecting classes must implement.

Returns: NasQos object containing the current QoS-values.

7.5.10 CMQosCollector

Package: crumpet.nas.ma
Syntax: public class CMQosCollector implements QosCollector
Direct Known Subclasses: none.
All Implemented Interfaces: none.

CMQosCollector provides QoS using congestion manager.

Constructors:
public CMQosCollector ()

Methods:
public NasQos getQos ()
This method uses the API of congestion manager to collect QoS. It does so through JNI interface.

Returns: NasQos object containing the current QoS-values.
7.5.11 The libcmmon library

The libcmmon library links the API calls from libcm to the Java CMQosCollector class through JNI. The same functionality as in libcm is provided for the CMQosCollector class. For the libcm API used in CRUMPET, see section 9.3 on page 98.

7.5.12 SimulatedQosCollector

Package: crumpet.nas.ma
Syntax: public class SimulatedQosCollector implements QosCollector
Direct Known Subclasses: none.
All Implemented Interfaces: none.

SimulatedQosCollector provides simulated QoS. The way this is done is out of scope of this document, but it can be by some simulator/emulator or by itself.

Constructors:

```
public SimulatedQosCollector()
```

Methods:

```
public NasQos getQos()
```

This method provides the simulated QoS.

Returns: NasQos object containing the current QoS-values.

7.5.13 NasQos

Package: crumpet.nas.ma
Syntax: public class NasQos
Direct Known Subclasses: none.
All Implemented Interfaces: none.

NasQos is a container for QoS information. This class carries the QoS values in its attributes. As this class does not implement any functionality, the attributes for holding the QoS values are public.

Constructors:

```
public NasQos()
```

Attributes:

- double lineRate
- String lineRateUnit
- double throughput
- String throughputUnit
- double throughputStdDev
- double rtt
- String rttUnit
- double rttStdDev
- double delay
- String delayUnit
- double delayStdDev
- boolean available
- String status
7.5.14 QosManager

*Package:* crumpet.nas.ma  
*Syntax:* public class QosManager  
*Direct Known Subclasses:* none.  
*All Implemented Interfaces:* none.

QosManager periodically collects QoS information using the QosCollector interface. The collected QoS information is stored into a persistent database using the QosHistory class and the standard deviations for the QoS values, of which the standard deviation are required, are calculated.

*Constructors:*

```java
public QosManager()
```

*Methods:*

```java
public void storeSubscription(AID aid, QosNotificationOntology ont)
```

This method stores a subscription.

*Parameters:*

aid — Agent identifier for the subscribed agent  
ont — The ontology object representing the subscription.

7.5.15 QosHistory

*Package:* crumpet.nas.ma  
*Syntax:* public class QosHistory  
*Direct Known Subclasses:* none.  
*All Implemented Interfaces:* none.

QosHistory keeps the last $n$ (where $n$ is user-configurable) QoS values in store. It allows adding NasQos objects to the history and is able to write the history to a persistent storage and vice versa, when the MonitorAgent is restarted, read the persistently stored QoS values into memory.

*Constructors:*

```java
public QosHistory(int n)
```

*Parameters:*

$n$ — the size of the history.

*Methods:*

```java
public void storeHistory()
```

This method stores the history to a persistent storage. This method is called at the shutdown of the MonitorAgent.

```java
public void readHistory()
```

This method reads the history from a persistent storage. This method is called at the initialisation of the MonitorAgent.

```java
public void addQos(NasQos qos)
```

This method appends a new QoS value to the history.

*Parameters:*

$qos$ — the QoS value to be added.
7.5.16 Subscriptions

**Package:** crumpet.nas.ma

**Syntax:** public interface Subscriptions

**Direct Known Subclasses:** none.

**All Implemented Interfaces:** none.

Subscriptions interface defines the functionality every concrete subscription database has to implement.

**Methods:**

- public boolean **addSubscription** (Subscription entry)
  
  This method adds a subscription.

  **Parameters:**
  
  entry — the new subscription entry.

  **Returns:** True, if the subscription was stored successfully. False otherwise.

- public boolean **removeSubscription** (Subscription entry)

  This method removes a subscription.

  **Parameters:**

  entry — the subscription entry to be removed.

  **Returns:** True, if the subscription was removed successfully. False otherwise.

- public Vector **matchSubscriptions** (NasQos qos)

  This method applies subscription matching.

  **Parameters:**

  qos — the QoS value which the matching is applied against.

  **Returns:** Vector of matched subscription entries.

7.5.17 SubscriptionStorage

**Package:** crumpet.nas.ma

**Syntax:** public class QosManager implements Subscriptions

**Direct Known Subclasses:** none.

**All Implemented Interfaces:** Subscriptions.

SubscriptionStorage implements the Subscriptions interface.

**Constructors:**

- public **SubscriptionStorage** ()

**Methods:**

- public boolean **addSubscription** (Subscription entry)

  This method adds a subscription.

  **Parameters:**

  entry — the new subscription entry.

  **Returns:** True, if the subscription was stored successfully. False otherwise.

- public boolean **removeSubscription** (Subscription entry)

  This method removes a subscription.

  **Parameters:**
entry — the subscription entry to be removed.

Returns: True, if the subscription was removed successfully. False otherwise.

public Vector matchSubscriptions (NasQos qos)
This method applies subscription matching.

Parameters:
qos — the QoS value which the matching is applied against.

Returns: Vector of matched subscription entries.

7.5.18 Subscription
Package: crumpet.nas.ma
Syntax: public interface Subscription
Direct Known Subclasses: none.
All Implemented Interfaces: none.

Subscription interface defines the functionality every concrete subscription entry has to implement.

Methods:
public boolean match (NasQos qos)
This method checks if the subscription matches against the QoS value passed as parameter.

Parameters:
qos — the QoS value which the matching is applied against.

Returns: True, if matched. False otherwise.

7.5.19 SubscriptionItem
Package: crumpet.nas.ma
Syntax: public class SubscriptionItem implements Subscription
Direct Known Subclasses: none.
All Implemented Interfaces: Subscription.

SubscriptionItem implements Subscription interface.

Constructors:
public SubscriptionItem (AID aid, QosNotificationOntology ont)

Parameters:
aid — subscriber agent's AID.
ont — QosNotification representing the subscription.

Methods:
public boolean match (NasQos qos)
This method checks if the subscription matches against the QoS value passed as parameter.

Parameters:
qos — the QoS value which the matching is applied against.

Returns: True, if matched. False otherwise.

Attributes:
AID aid
QosNotificationOntology ont
8 CONTROL AGENT

8.1 Description
The Control Agent (CA) is an agent, which controls the wireless message transport. Both the Mobile Device and the CRUMPET Access Node contain a CA, although the one in the Mobile Device is more relevant, because the Mobile Device initiates a session. The concrete functionality of the CA in CRUMPET includes:

- Establishing a message transport connection from the Mobile Device to the CRUMPET Access Node,
- Closing a message transport connection between the Mobile Device and the CRUMPET Access Node,
- Activating a message transport connection or message transport protocol,
- Deactivating a message transport connection or message transport protocol,

The CA has two parts: the FIPA-OS specific part is tightly coupled with FIPA-OS tasks while the platform-independent part is common for any platform.

8.2 External ACL interfaces

8.2.1 Open-comm-channel
An agent can request that a CA opens a message transport connection. The communication channel description should contain enough information for a CA to be able to choose the right communication channel, that is, either the :name parameter or the :target-addr parameter must be present. The agent also supplies additional communication channel information by using the :options parameter. Open-comm-channel follows the FIPA-Request [FIPA00i] interaction protocol, see the example\(^9\) in Figure 23.

![Figure 23: Open-comm-channel request.](image)

\(^9\) In this example not all the possible messages are taken into account. For a complete specification of the FIPA-Request interaction protocol, see [FIPA00i]. It must be noted that although this example uses SL2, there is no agreement in CRUMPET on which content language should be used. Furthermore, the string-based encoding used is just for clarity; at runtime messages are encoded bit-efficiently.
**(8.2.2) Close-comm-channel**

An agent can request that a CA closes a communication channel. The communication channel description should contain enough information for a CA to be able to choose the right communication channel, that is, either the :name parameter or the :target-addr parameter must be present.
Close-comm-channel follows the FIPA-Request [FIPA00i] interaction protocol, see the example in Figure 24.

![Figure 24: Close-comm-channel request.](image)

In this example not all the possible messages are taken into account. For a complete specification of the FIPA-Request interaction protocol, see [FIPA00i]. It must be noted that although this example uses SL2, there is no agreement in CRUMPET on which content language should be used. Furthermore, the string-based encoding used is just for clarity; at runtime messages are encoded bit-efficiently.
8.2.3 Activate

An agent can request that a CA activates a Message Transport Protocol (MTP). The transport protocol description should contain enough information to allow the CA to identify the correct transport protocol. Additionally, the agent may supply address information to where the transport protocol connection should be opened. It is possible to give the address of the gateway and/or the address of the destination AP. Activation follows the FIPA-Request [FIPA00i] interaction protocol, see the example\textsuperscript{11} in Figure 25.

\textbf{Figure 25: Activate request.}

\begin{verbatim}
[1] {request
  :sender
    {agent-identifier
      :name AgentA@foo.com}
  :receiver (set
    {agent-identifier
      :name CA@foo.com})
  :ontology FIPA-Nomadic-Application
  :language FIPA-SL0
  :protocol FIPA-Request
  :content
  {action
   {agent-identifier
     :name CA@foo.com}
   {activate (sequence
     {transport-protocol
      :name fipa.mts.mtp.wap.std
      :gw-addr wap://gateway.com/acc)})}
[2] {agree
  :sender
    {agent-identifier
      :name CA@foo.com}
  :receiver (set
    {agent-identifier
      :name AgentA@foo.com})
  :ontology FIPA-Nomadic-Application
  :language FIPA-SL0
  :protocol FIPA-Request
  :content
  {action
   ...
  }
\end{verbatim}

\textsuperscript{11} In this example not all the possible messages are taken into account. For a complete specification of the FIPA-Request interaction protocol, see [FIPA00i]. It must be noted that although this example uses SL2, there is no agreement in CRUMPET on which content language should be used. Furthermore, the string-based encoding used is just for clarity; at runtime messages are encoded bit-efficiently.
8.2.4 Deactivate

An agent can request that a CA deactivates an MTP. Deactivation follows the FIPA-Request interaction protocol, see the example\(^\text{12}\) in Figure 26.

---

\(^{12}\) In this example not all the possible messages are taken into account. For a complete specification of the FIPA-Request interaction protocol, see [FIPA00i]. It must be noted that although this example uses SL2, there is no agreement in CRUMPET on which content language should be used. Furthermore, the string-based encoding used is just for clarity; at runtime messages are encoded bit-efficiently.
8.3 Constraints and limitations

As with the MA, the main constraint for the design of the CA is the limited processing power and memory resources of the Mobile Device. Especially the CA in the Mobile Device has to be designed to be as lightweight as possible. One solution is not to have a full-blown CA extending the FIPAOSAgent class in the Mobile Device, but to design it so that the CA functionality is designed as a collection of FIPAOS tasks, which could be reused by agents in need of the CA’s services. As the CRUMPET Access Node does not expose such limitations as the Mobile Device, we can have a full-blown CA running there, utilising the same FIPAOS tasks as in the Mobile Device. As no prototypes are yet implemented on the runtime environment in the Mobile Device, we cannot say for sure, whether or not the CA in the Mobile Device should be left as a collection of FIPAOS tasks. However, we do not exclude this possibility from our design.

The amount of threads may cause performance problems, especially if every task runs in its own thread. Therefore, we have to take into account that the number of tasks and threads may have to be reduced to achieve better performance. The design for the Control Agent allows for a reduction of the amount of threads used. As soon as the first prototype of the MicroFIPA-OS can be tested for performance, we have a better understanding about the effects of threading on the performance.

Also, analogously with the constraint of MA, the second constraint is the dependency on the operating system in the implementation of the actual controlling functions for the MTC, because Java does not
allow access to ports and devices. Therefore, as with the MA, the design divides into "two layers". The "upper layer" is a common one defining the interfaces, which the "lower level" implementation-specific components must implement. Whilst the upper layer is designed to be compatible with any Java enabled device, the lower level depends upon the operating system to be used. In CRUMPET, the operating system to be used is LINUX and the lower layer controlling functions are LINUX-specific functions.

8.4 Class diagrams

8.4.1 FIPA-OS specific classes

The classes implementing the FIPAOS task of the CA are depicted in Figure 27. In the CRUMPET Access Node, all the functionality is encompassed in the ControlAgent, which is inherited from the FIPAOSAgent thereby making it a FIPAOS agent. The ControlAgent includes a number of tasks, which are inherited from the FIPAOS task allowing the FIPAOS task manager to manage them. If the limitations of the Mobile Device are too restrictive in processing power and the amount of runtime memory, the ControlAgent will not be implemented in the Mobile Device, but instead the tasks will be allowed to be adopted by other (client) agents directly. The ControlAgent's tasks utilise the CAFunctions class to execute the control agent specific functions. The CAFunction mediates the FIPA-OS specific and platform-independent parts.

The CAFunctions class provides a single entry point for accessing the CA services. Thus, it acts as a facade [Gra98] for the underlying CA functionality. The CA tasks do not have to know the details of how an incoming request in ACL message is actually handled; all they have to know is the interface of the CAFunctions for calling the execution method with the NasOntology object as parameter. In return, upon completing the request, the MAFunctions class returns the result NasOntology object to the CA task.

The CAFunctions class is a singleton class [Gra98]. Therefore, it contains a static method to get a reference to it thus relieving the CA tasks from the need to know how the CAFunctions class should be instantiated. Singleton pattern also assures that there will not be more than one instance of the CAFunctions.

The CAQueryListenerTask listens for incoming messages targeted for CA. Based on the message, CAQueryListenerTask creates an appropriate CA-specific task to take care of message. These CA-specific classes (OpenCommChannelTask, CloseCommChannelTask, ActivateTask and DeactivateTask) in turn create NasOntology object out of the message (using NasOntologyFactory) and delegate the handling of the NasOntology to CAFunctions class.

The classes in Figure 27 are specified in more detail in upcoming section.
8.4.2 Platform-independent classes

In Figure 28 the platform-independent classes of CA are depicted. Based on the type of request, the CAFunctions delegates the handling of the request to an appropriated handler class, which in control agent's case are the handlers for open-comm-channel, close-comm-channel, activate and deactivate requests.

OpenCommChannelFunction, CloseCommChannelFunction, ActivateFunction and DeactivateFunction implement the actual functions. All of them implement NasFunction interface therefore defining execute-function, which the CAFunctions-object calls with the NasOntology object as parameter. Thus, CAFunctions object only needs to construct an appropriate function class for executing the request implicitly defined by the NasOntology object. This allows strategy pattern-like [Gra98] behaviour, where the CAFunctions object has a set of different strategies for handling the incoming request. This also promotes scalability and reuse, because the function classes can be added and replaced at runtime.

The OpenCommChannelFunction is responsible to open a communication channel to the CRUMPET Access Node. It delegates the actual connection setup to ConnectionManager. The CloseCommChannel in turn uses the ConnectionManager to close a connection.

The ActivateFunction is responsible to activate a MTP, whereas the DeactivateFunction takes care of deactivating one.

The DeviceHandler class is able to listen to events from both PCMCIA slots and serial ports. It aggregates two specialised objects, PCMCIAHandler and SerialHandler. The PCMCIAHandler is specialised in the events from the PCMCIA slots whereas the SerialHandler polls the serial ports and informs about device connected/disconnected to the serial port. The DeviceHandler implements the DeviceNotifier interface, which allows DeviceListeners to be registered as listeners and therefore be notified by the DeviceHandler upon an event. The design follows observer pattern [Gra98].

The classes in Figure 28 are specified in more detail in upcoming section.
8.4.3 CA profiles
Profiles divide into two logically separate parts: network device profiles and user preferences. Network device profiles contain static information about the characteristics of the network devices. User preferences contain dynamic information about users' preferences on using the wireless link. In Figure 29 the classes for the profiles are presented.

The ConnectionManager has a central role in both profile and connection handling. It acts as a Factory [Gra98] creating NetworkDevice objects based on the device ID. The ConnectionManager is also able to return the list of currently attached devices as a list. Therefore, two possible situations are covered. A device is attached to the CRUMPET Client and user preferences are browsed through to see if there exists user wanting to use the device. The other possibility is that the device is already attached and user wants to use the device.

Figure 28: Platform-independent control agent classes.
When either PCMCIA or serial device is attached, DeviceHandler notifies ConnectionManager with device ID as parameter. ConnectionManager creates a NetworkDevice object based on the device ID. The status of the NetworkDevice is set "uninitialised". ConnectionManager initialises the NetworkDevice and after successful initialisation, sets the status to "initialised". After this, the AccessPoint profile is checked if there exists such an Access Point, which the device can connect to. If such was found, the user preferences are iterated through to see if there is a user, who wants to use the device to connect to the Access Point. Using LoginInfo and AccessPoint as parameters, the ConnectionManager uses appropriate connector class to make the connection.

In the other case, when the user is initiating the connection setup, OpenCommChannel first asks ConnectionManager for the currently attached devices. If the device is attached, it is initialised, if needed. After this, the user preferences are checked for the login-specific information and the AccessPoint to be connected to. After this, appropriate connector class sets up the connection.

Both network device and user preference information must be configured by the administrator/user. Also, both of them are of persistent type. Therefore, they are stored onto persistent database when the CRUMPET Client is shutdown\(^\text{13}\) and read into the memory at the start-up. In the following, the database objects and their fields are described. In the case of dual type devices, i.e. the devices providing multiple network interfaces, one network device frame is created for each network interface.

\(^\text{13}\) I.e. the CA is shutdown.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Type</th>
<th>Reserved values</th>
</tr>
</thead>
<tbody>
<tr>
<td>deviceID</td>
<td>Unique ID for the device.</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name of the device.</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>Type of the device.</td>
<td>String</td>
<td>gsm hscsd wlan gprs umts lan</td>
</tr>
<tr>
<td>linerate-inbound</td>
<td>Inbound linerate for the device.</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>linerate-outbound</td>
<td>Outbound linerate for the device.</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>properties</td>
<td>Device-specific properties. For examples of these properties, see Table 12.</td>
<td>Set of Property</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROPERTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACCESS POINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
</tr>
<tr>
<td>apID</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>type</td>
</tr>
<tr>
<td>owner</td>
</tr>
<tr>
<td>properties</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>USER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
</tr>
<tr>
<td>username</td>
</tr>
<tr>
<td>LoginInfos</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOGIN INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
</tr>
<tr>
<td>userID</td>
</tr>
<tr>
<td>passwd</td>
</tr>
<tr>
<td>apID</td>
</tr>
</tbody>
</table>

Table 12 introduces the network properties available on both the Mobile Device and Access Point (“Device” and “AP” columns, respectively).
### GSM-specific properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>phone-number</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>init-string</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>data-bits</td>
<td>Integer</td>
<td>4, 5, 6, 7, 8</td>
</tr>
<tr>
<td>parity</td>
<td>String</td>
<td>even, odd, none</td>
</tr>
<tr>
<td>stop-bits</td>
<td>Integer</td>
<td>1, 2</td>
</tr>
<tr>
<td>connection-method</td>
<td>String</td>
<td>modem, isdn_110, isdn_120</td>
</tr>
<tr>
<td>linerate-inbound</td>
<td>String</td>
<td>9.6, 14.4, 19.2, 28.8, 43.2</td>
</tr>
<tr>
<td>linerate-outbound</td>
<td>String</td>
<td>9.6, 14.4, 19.2, 28.8, 43.2</td>
</tr>
</tbody>
</table>

### WLAN-specific properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>network-name</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>ap-name</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>channel</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>encryption</td>
<td>String</td>
<td>on, off</td>
</tr>
<tr>
<td>connection-method</td>
<td>String</td>
<td>ap, adhoc</td>
</tr>
</tbody>
</table>

### GPRS-specific properties

See (*)

### UMTS-specific properties

See (*)

### TCP/IP properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip-address</td>
<td>String</td>
<td>dhcp</td>
</tr>
<tr>
<td>dns-address</td>
<td>String</td>
<td>dhcp</td>
</tr>
<tr>
<td>netmask</td>
<td>String</td>
<td>dhcp</td>
</tr>
<tr>
<td>broadcast-address</td>
<td>String</td>
<td>dhcp</td>
</tr>
<tr>
<td>default-gw</td>
<td>String</td>
<td>dhcp</td>
</tr>
</tbody>
</table>

Table 12: Properties.

(*) At the time of writing this Deliverable, the GPRS and UMTS properties are not known. Therefore, they will be added as soon as we gain more knowledge about GPRS and UMTS devices.

Information on the persistent database is formatted according to the format CRUMPET chooses to use for other profiles also. As the CRUMPET-wide format is not decided yet, an example of network device database in RDF format is shown in Figure 30.
<?xml version="1.0"?>
<rdf:RDF xml:lang="en"
  xmlns="http://foo.com/schemas/devices#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:nas="http://foo.com/schemas/devices#">
  <nas:NetworkDevice rdf:about="cp2">
    <nas:deviceID> nokia_cp2 </nas:deviceID>
    <nas:name> Nokia CardPhone 2.0 </nas:name>
    <nas:type> gsm </nas:type>
    <nas:linerate> 47.6 kbit/s </nas:linerate>
    <nas:properties>
      <nas:property>
        <nas:name> init_string </nas:name>
        <nas:value> at+cbst=51,0,1;+chsn=6,0,0,0 </nas:value>
      </nas:property>
    </nas:properties>
  </nas:NetworkDevice>

  <nas:AccessPoint rdf:about="crumpet.sonera.fi">
    <nas:apID> gsm@crumpet.sonera.fi </nas:apID>
    <nas:name> crumpet.sonera.fi </nas:name>
    <nas:owner> sonera </nas:owner>
    <nas:type> gsm </nas:type>
    <nas:properties>
      <nas:property>
        <nas:name> phone_number </nas:name>
        <nas:value> +358 0 1234567 </nas:value>
      </nas:property>
      <nas:property>
        <nas:name> data_bits </nas:name>
        <nas:value> 8 </nas:value>
      </nas:property>
      <nas:property>
        <nas:name> parity </nas:name>
        <nas:value> none </nas:value>
      </nas:property>
      <nas:property>
        <nas:name> stop_bits </nas:name>
        <nas:value> 1 </nas:value>
      </nas:property>
      <nas:property>
        <nas:name> connection_method </nas:name>
        <nas:value> isdn_120 </nas:value>
      </nas:property>
      <nas:property>
        <nas:name> speed </nas:name>
        <nas:value> 43.2 </nas:value>
      </nas:property>
      <nas:property>
        <nas:name> ip_address </nas:name>
        <nas:value> 10.0.0.21 </nas:value>
      </nas:property>
      <nas:property>
        <nas:name> dns_address </nas:name>
        <nas:value> 10.0.0.1 </nas:value>
      </nas:property>
    </nas:properties>
  </nas:AccessPoint>

  <nas:User rdf:about="mikkolaukkanen">
    <nas:username> Mikko Laukkanen </nas:username>
    <nas:logininfo>
      <nas:userID> mikko </nas:userID>
      <nas:passwd> crumpet </nas:passwd>
      <nas:apID> gsm@crumpet.sonera.fi </nas:apID>
    </nas:logininfo>
  </nas:User>
</rdf:RDF>

Figure 30: Profile in RDF format.
8.5 Class definitions

8.5.1 ControlAgent

Package: crumpet.nas.ca
Syntax: public class ControlAgent extends FIPAOSAgent
Direct Known Subclasses: none.
All Implemented Interfaces: none.

ControlAgent encapsulates the CA functionality in a FIPA-OS agent. This class encapsulates all the CA functionality and takes care of registering with AMS and DF.

Constructors:
public ControlAgent (String platform, String name, String owner)

8.5.2 CAQueryListenerTask

Package: crumpet.nas.ca
Syntax: public class CAQueryListenerTask extends Task
Direct Known Subclasses: none.
All Implemented Interfaces: none.

CAQueryListenerTask listens for incoming requests from other agents. CAQueryListenerTask is FIPAOS task, which listens for incoming requests from other agents and upon receiving one, dispatches the appropriate task for taking care of handling the request.

Constructors:
public CAQueryListenerTask ()

Methods:
public void handleRequest (Conversation conv)
   This method handles the incoming request.

   Parameters: conv — the conversation to which the request belongs.
   CARequestListererTask should ask the Conversation object for the latest message in order to get the request message.

8.5.3 OpenCommChannelTask

Package: crumpet.nas.ca
Syntax: public class OpenCommChannelTask extends Task
Direct Known Subclasses: none.
All Implemented Interfaces: none.

OpenCommChannelTask All Implemented Interfaces: the open-comm-channel functionality. This class is responsible for handling the open-comm-channel query, but delegates most of the functionality to the CAFunctions class. This design decision promotes portability between different FIPA compliant agent platforms. So basically this class hooks the open-comm-channel functionality in FIPA-OS. First, agree message is sent to the requesting agent. Then the actual open-comm-channel query is delegated to CAFunctions and after it has returned with the result NasOntology, inform message is sent to the requesting agent.

Constructors:
public OpenCommChannelTask ()

Methods:
public void openCommChannel (ACL acl)
   This method is responsible for opening a communication channel. It does so by delegating the responsibility for CAFunctions class.
Parameters: acl — The ACL message containing the open-comm-channel function as content.

8.5.4 CloseCommChannelTask

Package: crumpet.nas.ca
Syntax: public class CloseCommChannelTask extends Task
Direct Known Subclasses: none.
All Implemented Interfaces: none.

CloseCommChannelTask All Implemented Interfaces: the close-comm-channel functionality. This class is responsible for handling the close-comm-channel query, but delegates most of the functionality to the CAFunctions class. This design decision promotes portability between different FIPA compliant agent platforms. So basically this class hooks the close-comm-channel functionality in FIPA-OS. First, agree message is sent to the requesting agent. Then the actual close-comm-channel query is delegated to CAFunctions and after it has returned with the result NasOntology, inform message is sent to the requesting agent.

Constructors:
public CloseCommChannelTask ()

Methods:
public void closeCommChannel (ACL acl)
This method is responsible for closing a communication channel. It does so by delegating the responsibility for CAFunctions class.
Parameters: acl — The ACL message containing the close-comm-channel function as content.

8.5.5 ActivateTask

Package: crumpet.nas.ca
Syntax: public class ActivateTask extends Task
Direct Known Subclasses: none.
All Implemented Interfaces: none.

ActivateTask All Implemented Interfaces: the activate functionality. This class is responsible for handling the activate query, but delegates most of the functionality to the CAFunctions class. This design decision promotes portability between different FIPA compliant agent platforms. So basically this class hooks the activate functionality in FIPA-OS. First, agree message is sent to the requesting agent. Then the actual activate query is delegated to CAFunctions and after it has returned with the result NasOntology, inform message is sent to the requesting agent.

Constructors:
public ActivateTask ()

Methods:
public void activate (ACL acl)
This method is responsible for activating an MTP. It does so by delegating the responsibility for CAFunctions class.
Parameters: acl — The ACL message containing the activate function as content.
8.5.6 DeactivateTask

Package: crumpet.nas.ca
Syntax: public class DeactivateTask extends Task
Direct Known Subclasses: none.
All Implemented Interfaces: none.

DeactivateTask All Implemented Interfaces: the deactivate functionality. This class is responsible for handling the deactivate query, but delegates most of the functionality to the CAFunctions class. This design decision promotes portability between different FIPA compliant agent platforms. So basically this class hooks the deactivate functionality in FIPA-OS. First, agree message is sent to the requesting agent. Then the actual deactivate query is delegated to CAFunctions and after it has returned with the result NasOntology, inform message is sent to the requesting agent.

Constructors:
public DeactivateTask ()

Methods:
public void deactivate (ACL acl)
This method is responsible for deactivating an MTP. It does so by delegating the responsibility for CAFunctions class.

Parameters: acl — The ACL message containing the deactivate function as content.

8.5.7 DeviceListener

Package: crumpet.nas.ca
Syntax: public interface DeviceListener
Direct Known Subclasses: none.
All Implemented Interfaces: none.

DeviceListener interface must be implemented by all the classes, which want to receive notifications about events occurring on both PCMCIA slots and serial port.

Methods:
public void notify (DeviceEvent event)
DeviceNotifiers call this method in order to tell DeviceListener implementers that an event has occurred.

Parameters: event — The DeviceEvent object representing the event occurred.

8.5.8 DeviceEvent

Package: crumpet.nas.ca
Syntax: public class DeviceEvent
Direct Known Subclasses: none.
All Implemented Interfaces: none.

DeviceEvent encapsulates the event-related information. It contains the type (i.e. the source of the event) as well as the device-specific device ID.

Attributes:
int type (is either DeviceEvent.PCMCIA or DeviceEvent.SERIAL)
String deviceId
8.5.9 DeviceNotifier

Package: crumpet.nas.ca
Syntax: public interface DeviceNotifier
Direct Known Subclasses: none.
All Implemented Interfaces: none.

DeviceNotifier interface must be implemented by the classes, which monitor the PCMCIA slots or serial port. This interface allows registration and deregistration for the notification about events. The methods defined by this interface are called by the EventListeners.

Methods:

public void addListener (DeviceListener listener)
By calling this method a DeviceListener is registered for receiving notifications about events occurring on both PCMCIA slots and serial port.

Parameters: listener — The object who registers for notifications.

public void removeListener (DeviceListener listener)
By calling this method a DeviceListener deregisters from receiving notifications about events occurring on both PCMCIA slots and serial port.

Parameters: listener — The object who deregisters from notifications.

8.5.10 DeviceHandler

Package: crumpet.nas.ca
Syntax: public class DeviceHandler implements DeviceNotifier
Direct Known Subclasses: none.
All Implemented Interfaces: DeviceNotifier.

DeviceHandler is aggregate class for the actual DeviceHandlers, which monitor the PCMCIA slots and serial port. DeviceHandler is responsible to handle registrations and deregistrations and to route the events from the actual DeviceHandlers to the registered DeviceListeners. Therefore, DeviceHandler implements the DeviceNotifier interface.

Constructors:

public DeviceHandler ()

Methods:

public void addListener (DeviceListener listener)
By calling this method a DeviceListener is registered for receiving notifications about events occurring on both PCMCIA slots and serial port.

Parameters: listener — The object who registers for notifications.

public void removeListener (DeviceListener listener)
By calling this method a DeviceListener deregisters from receiving notifications about events occurring on both PCMCIA slots and serial port.

Parameters: listener — The object who deregisters from notifications.

public void dispatchEvent (DeviceEvent event)
This method is called by the actual DeviceHandler classes (PCMCIAHandler or SerialHandler). This method in turn distributes the received event to all the DeviceListeners.
Parameters: event — The even occurred.

8.5.11 PCMCIAHandler

Package: crumpet.nas.ca
Syntax: public class PCMCIAHandler
Direct Known Subclasses: none.
All Implemented Interfaces: none.

PCMCIAHandler is able to monitor the PCMCIA slots and inform DeviceHandler about events occurring in them. Upon event, the DeviceEvent is constructed and passed to the DeviceHandler. PCMCIAHandler does not have any public methods. It acts autonomously and is only responsible to inform about PCMCIA events.

Constructors:
public PCMCIAHandler (DeviceHandler parent)
Parameters: parent — A reference to the aggregating DeviceHandler class, to which the events are supposed to be sent.

8.5.12 SerialHandler

Package: crumpet.nas.ca
Syntax: public class SerialHandler
Direct Known Subclasses: none.
All Implemented Interfaces: none.

SerialHandler is able to monitor the serial port and inform DeviceHandler about events occurring in it. Upon event, the DeviceEvent is constructed and passed to the DeviceHandler. SerialHandler does not have any public methods. It acts autonomously and is only responsible to inform about serial port events.

Constructors:
public SerialHandler (DeviceHandler parent)
Parameters: parent — A reference to the aggregating DeviceHandler class, to which the events are supposed to be sent.

8.5.13 CAFunctions

Package: crumpet.nas.ca
Syntax: public class CAFunctions
Direct Known Subclasses: none.
All Implemented Interfaces: none.

CAFunctions acts as a facade assigning responsibility to CA functions classes based on the incoming request. CAFunctions class is a singleton class implementing one common function for executing CA functions. It does so by first creating the appropriate object and then assigning the work for to the object.

Constructors:
public CAFunctions ()

Methods:
public NasOntology execute (NasOntology ontology)
This method is called with NasOntology as parameter. Based on the type of the NasOntology object, one of the CA function classes is created and the responsibility of executing the function is passed to it.

Parameters: ontology — The ontology object representing the function.
**8.5.14 OpenCommChannelFunction**

*Package:* crumpet.nas

*Syntax:* public class OpenCommChannelFunction extends NasFunction

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* NasFunction.

OpenCommChannelFunction is able to open a peer-to-peer connection. This class opens a connection to a peer agent platform. ConnectionManager class is used for the actual connection setup. After the connection is set up, a NasOntology object is returned with content indicating about the success of the function.

*Constructors:*

```java
public OpenCommChannelFunction()
```

*Methods:*

```java
public NasOntology execute(NasOntology ontology)
```

This method implements the opening of the communication channel.

*Parameters:* ontology — The ontology object representing the open-comm-channel function.

*Returns:* New ontology object representing the result of the executed function.

**8.5.15 CloseCommChannelFunction**

*Package:* crumpet.nas

*Syntax:* public class CloseCommChannelFunction extends NasFunction

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* NasFunction.

CloseCommChannelFunction is able to close a peer-to-peer connection. This class closes a connection to a peer agent platform. ConnectionManager class is used for the actual connection setup. After the connection is closed, a NasOntology object is returned with content indicating about the success of the function.

*Constructors:*

```java
public CloseCommChannelFunction()
```

*Methods:*

```java
public NasOntology execute(NasOntology ontology)
```

This method implements the closing of the communication channel.

*Parameters:* ontology — The ontology object representing the close-comm-channel function.

*Returns:* New ontology object representing the result of the executed function.

**8.5.16 ActivateFunction**

*Package:* crumpet.nas

*Syntax:* public class ActivateFunction extends NasFunction

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* NasFunction.
This class activates an MTP. After the MTP has been activated, a NasOntology object is returned with the content indicating about the success of the function.

**Constructors:**

```java
public ActivateFunction ()
```

**Methods:**

```java
public NasOntology execute (NasOntology ontology)
```

This method implements the activation of MTP.

*Parameters:* ontology — The ontology object representing the activate function.

*Returns:* New ontology object representing the result of the executed function.

---

**8.5.17 DeactivateFunction**

*Package:* crumpet.nas.ca

*Syntax:* public class DeactivateFunction implements NasFunction

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* NasFunction.

This class deactivates an MTP. After the MTP has been deactivated, a NasOntology object is returned with the content indicating about the success of the function.

**Constructors:**

```java
public DeactivateFunction ()
```

**Methods:**

```java
public NasOntology execute (NasOntology ontology)
```

This method implements the deactivation of MTP.

*Parameters:* ontology — The ontology object representing the deactivate function.

*Returns:* New ontology object representing the result of the executed function.

---

**8.5.18 NasFunction**

For definition, see section 7.5.6 on page 64.

---

**8.5.19 NetworkDevice**

*Package:* crumpet.nas.ca

*Syntax:* public class NetworkDevice

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* none.

NetworkDevice represents a device capable of providing network connection.

**Constructors:**

```java
public NetworkDevice (String deviceID, String name, String type, double linerate, Vector properties)
```

*Parameters:*

- deviceID — The deviceID assigned for the device.
- name — Name of the device.
- type — Type of the device.
- linerate — The theoretical linerate for the device.
properties — A set of device-specific properties.

    public NetworkDevice ()
    Empty constructor. All the attributes are initialised as null.

*Methods:*

    public boolean initialise ()
    Initialises the device. Every device has its unique initialisation procedure, which is
    implemented by this method. After successful initialisation, the status for the device
    is set to "initialised".

    *Returns:* True, if device was successfully initialised. False otherwise.

*Attributes:*

    String deviceID
    String name
    String type
    double linerate
    String status
    Vector properties

### 8.5.20 ConnectionManager

**Package:** crumpet.nas.ca

**Syntax:** public class ConnectionManager implements DeviceListener

**Direct Known Subclasses:** none.

**All Implemented Interfaces:** DeviceListener.

ConnectionManager orchestrates the connections. It provides a factory for the devices so that it is able
to create NetworkDevice on request based on a unique device ID, which is device-specific. By
implementing the DeviceListener interface it receives notifications about PCMCIA slots and serial
port.

*Constructors:*

    public ConnectionManager ()

*Methods:*

    public NetworkDevice addDevice (String deviceID)
    Adds a new NetworkDevice to the list of attached devices.

    *Parameters:* deviceID — A device specific ID.

    *Returns:* The created NetworkDevice object.

    public NetworkDevice createDevice (String deviceID)
    Creates a new NetworkDevice based on device ID.

    *Parameters:* deviceID — A device specific ID.

    *Returns:* The created NetworkDevice object.

    public boolean isAttached (String deviceID)
    Checks if the device with the deviceID is attached.

    *Parameters:* deviceID — A device specific ID.

    *Returns:* True, if the device is attached. False otherwise.

    public Vector getAttachedDevices ()
Returns a list of all the currently attached devices.

*Returns:* a vector containing the attached NetworkDevice's.

```java
public void notify(DeviceEvent ev)
```
Is called to notify about event occurred either in PCMCIA slot or serial port.

*Parameters:* `ev` — The event.

### 8.5.21 Connector

*Package:* crumpet.nas.ca

*Syntax:* public interface Connector

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* none.

The connector interface must be implemented by all the classes being able to make a connection to the CRUMPET Access Point.

*Methods:*

```java
public boolean connect(AccessPoint ap, LoginInfo li)
```
Makes the connection to the CRUMPET Access Point using the information in LoginInfo.

*Parameters:*

- `ap` — The CRUMPET Access Point, to which the connection is made.
- `li` — The LoginInfo to be used while authenticating.

*Returns:* True, if the connection was made successfully. False otherwise.

### 8.5.22 GSMConnector

*Package:* crumpet.nas.ca

*Syntax:* public class GSMConnector implements Connector

*Direct Known Subclasses:* none.

*All Implemented Interfaces:* none.

This class is able to make GSM connection to the CRUMPET Access Node.

*Constructors:*

```java
public GSMConnector ()
```

*Methods:*

```java
public boolean connect(AccessPoint ap, LoginInfo li)
```
Makes the connection to the CRUMPET Access Point using the information in LoginInfo.

*Parameters:*

- `ap` — The CRUMPET Access Point, to which the connection is made.
- `li` — The LoginInfo to be used while authenticating.

*Returns:* True, if the connection was made successfully. False otherwise.
8.5.23 GPRSConnector

Package: crumpet.nas.ca
Syntax: public class GPRSConnector implements Connector
Direct Known Subclasses: none.
All Implemented Interfaces: none.

This class is able to make GPRS connection to the CRUMPET Access Node.

Constructors:
  public GPRSConnector ()

Methods:
  public boolean connect (AccessPoint ap, LoginInfo li)
  Makes the connection to the CRUMPET Access Point using the information in LoginInfo.

Parameters:
  ap — The CRUMPET Access Point, to which the connection is made.
  li — The LoginInfo to be used while authenticating.

Returns: True, if the connection was made successfully. False otherwise.

8.5.24 WLANConnector

Package: crumpet.nas.ca
Syntax: public class WLANConnector implements Connector
Direct Known Subclasses: none.
All Implemented Interfaces: none.

This class is able to set up WLAN connection to the CRUMPET Access Node.

Constructors:
  public WLANConnector ()

Methods:
  public boolean connect (AccessPoint ap, LoginInfo li)
  Makes the connection to the CRUMPET Access Point using the information in LoginInfo.

Parameters:
  ap — The CRUMPET Access Point, to which the connection is made.
  li — The LoginInfo to be used while authenticating.

Returns: True, if the connection was made successfully. False otherwise.

8.5.25 UserProfile

Package: crumpet.nas.ca
Syntax: public class UserProfile
Direct Known Subclasses: none.
All Implemented Interfaces: none.

UserProfile is a collection of User-objects.
Constructors:

public UserProfile ()

Methods:

public UserProfile readProfile ()
Reads the profile from the persistent storage.

Returns: The constructed UserProfile object.

public boolean storeProfile (String filename)
Stores the profile onto the persistent storage.

Parameters:

filename — The name of the file to which the UserProfile is stored.

Returns: True, if the UserProfile was successfully stored. False otherwise.

public Vector getLoginInfos (String apID)
Gets all the LoginInfo objects for the CRUMPET Access Point, who's ID is passed as a parameter.

Parameters: apID — The ID for the CRUMPET Access Node.

Returns: a vector of LoginInfo objects.

8.5.26 User
Package: crumpet.nas.ca
Syntax: public class User
Direct Known Subclasses: none.
All Implemented Interfaces: none.

User object represents a user. It aggregates one or more LoginInfo objects.

Constructors:

public User ()

Methods:

public LoginInfo getLoginInfo (String apID)
Returns user's LoginInfo object for the AccessPoint, whose apID is passed as parameter.

Parameters: apID — The AccessPoint apID.

Returns: The LoginInfo for the AccessPoint in question.

Attributes:

String name

8.5.27 LoginInfo
Package: crumpet.nas.ca
Syntax: public class LoginInfo
Direct Known Subclasses: none.
All Implemented Interfaces: none.

LoginInfo encapsulates information needed when logging onto the CRUMPET Access Node. This information covers userID and password.
Constructors:

public LoginInfo (String userID, String passwd, String apID)

Parameters:

userID — A valid userID for the CRUMPET Access Node login.
passwd — A valid password for the CRUMPET Access Node login.
apID — The ID for the CRUMPET Access Node to which this LoginInfo is valid for.

Attributes:

String userID
String passwd
String apID

8.5.28 AccessPointProfile

Package: crumpet.nas.ca
Syntax: public class AccessPointProfile
Direct Known Subclasses: none.
All Implemented Interfaces: none.

AccessPointProfile is a collection of AccessPoint objects.

Constructors:

public AccessPointProfile ()

Methods:

public UserProfile readProfile ()
Reads the profile from the persistent storage.

Returns: The constructed UserProfile object.

public boolean storeProfile (String filename)
Stores the profile onto the persistent storage.

Parameters: filename — The name of the file to which the UserProfile is stored.

Returns: True, if the UserProfile was successfully stored. False otherwise.

public Vector getAccessPoints (NetworkDevice device)
Returns all the AccessPoints to which the NetworkDevice passed as parameter is capable of connecting to.

Parameters: device — the Network device.

Returns: a vector of AccessPoint objects to which the NetworkDevice is able to connect.

8.5.29 AccessPoint

Package: crumpet.nas.ca
Syntax: public class AccessPoint
Direct Known Subclasses: none.
All Implemented Interfaces: none.
The AccessPoint represents all the information about the CRUMPET Access Node's access point that is needed for uniquely identify and successfully log onto the CRUMPET Access Node using the access point.

**Constructors:**

```java
public AccessPoint ()
```

**Attributes:**

- String `apID`
- String `name`
- String `type`
- String `owner`
- Vector `properties`
9 CONGESTION MANAGER

9.1 Description
The CM is an end-to-end framework for congestion control and management, bandwidth sharing, independent of specific transport protocols (like TCP) and applications [And00, Bal99]. Its end-system architecture enables logically different flows (such as multiple concurrent Web downloads, concurrent audio and video streams, etc.) to adapt to congestion, share network information, and share (varying) available bandwidth well. Rather than have each stream act in isolation and thereby adversely interact with the others, the CM maintains host- and domain-specific path information, and orchestrates all transmissions. The CM’s internal algorithms ensure social and stable network behaviour; its API enables a variety of applications and transport protocols to adapt to congestion and varying bandwidth. The CM architecture is presented in Figure 31.

![Figure 31: The CM architecture [CM01].](image)

The CM maintains network statistics across flows, orchestrates data transmissions governed by robust control principles, and obtains feedback from the receiver, using a Congestion Controller, Flow Scheduler, and Feedback Prober. It also exports a simple yet powerful API for applications to learn about network state and adapt their data transmissions to obtain the best possible performance. We are also exploring several new algorithms for end-to-end congestion control, especially for streaming real-time audio and video. The CM framework provides a modular implementation platform for these ideas.

The CM consists of two "parts": the kernel part and the CM library, which is accessed by user API. Control sockets handle the communication between these. The kernel part replaces the LINUX TCP congestion algorithms by CM functionality. The CM library provides user with an API to learn about and adapt to network congestion and varying bandwidth. The interaction between the kernel part and the CM library is outlined in Figure 32.
Applications wanting to use CM controlled network connections must call `cm_open()`, which opens a control socket to the CM and associates the source and destination addresses (passed as parameters) with the CM macroflow. The return value is a handle, which the user uses in future CM calls. When the application terminates the connection, it calls the `cm_close()` to close the control socket to CM and to let the CM remove the association to the corresponding macroflow.

There are three ways to use CM-enabled data transmission:

- **Buffered send**: the conventional `write()` or `sendto()` calls are used, but CM paces the transmission.
- **Request/callback**: client doesn't send data itself, but registers a data sending function, which the CM calls, when it permits the client to send. The client is permitted to send up to MTU bytes of data.
- **Rate callback**: a client sending data on fixed schedule may receive callbacks from CM notifying about changes in parameters of the communication channel. Using this knowledge, the client can adjust its timer loops.

As the CM requires no changes at the receiver, the senders are required to give feedback about transmission. Therefore, senders must inform CM about the number of sent and received packets, type of congestion loss (if any) and a round-trip time. These values are delivered to CM using `cm_update()` function.

Clients are also able to learn about the current bandwidth and round-trip time by calling the `cm_query()` function. This is especially useful in CRUMPET, because the characteristics of the wireless link change constantly. This information can be used in controlling the link as well as applying content adaptation.

The CM is implemented in LINUX kernel and both TCP and UDP are implemented as in-kernel clients for CM. All the LINUX TCP congestion algorithms are replaced by CM algorithms, some CM data structures are added and the control socket facility for the user API is provided. TCP uses the request/callback mechanism for transmitting data. UDP on the other hand uses the buffered send mechanism.
9.2 Constraints and limitations
Congestion Manager replaces functionality inside LINUX kernel, which makes it tightly coupled with the kernel version used. Therefore, the main constraint is the portability to other kernels.

9.3 Interfaces
CM provides an API to access its services. This API is provided by libcm library. The API calls used in CRUMPET are listed in the following (for complete list, see [CM01]):

```c
int cm_open(struct sockaddr_in *srcaddr, struct sockaddr_in *dstaddr)
```
cm_open creates (and returns) a new macroflow. It takes as its arguments pointers to the sockaddr_in structures of the source and the destination of the connection (or the stream, aka flow in this document). On error -1 is returned, otherwise 0 is returned.

```c
int cm_mtu(int id)
```
Returns the MTU value (in bytes) for a macroflow. It takes the macroflow as its argument.

```c
int cm_request(int id)
```
Used by an ALF (Application Level Framing) application to express its desire to send data. Takes macroflow as its argument and returns the status. If the request was scheduled correctly, 0 is returned. Otherwise -1 is returned. Calling cm_request on an already scheduled flow also returns -1.

```c
int cm_update(int id, int nrecd, int nlost, int lossmode, unsigned int rtt)
```
Used to provide feedback to the CM about the amount of data received or lost for the stream and/or the round-trip time (RTT) value for this flow. Takes a macroflow as its argument. In case of a TCP connection using the CM, the in-kernel TCP uses acknowledgements to do cm_update(). Hence, cm_update() needs to be called only if the user-level application is providing feedback (in particular, for non-TCP-based applications). lossmode can be one of the following: CM_NOLOSS, CM_PERSISTENT, CM_TRANSIENT or CM_ECN. These correspond to no congestion losses, "persistent" losses that cause the stream to lose its TCP-like self-clock, "transient" congestion where the self-clock isn't lost although congestion has occurred, and congestion involving no losses but the use of explicit congestion notification.

```c
int cm_query(int id, int *ready, cmquery *cmquery)
```
Used by an application to get statistics for a flow from the CM. Takes a macroflow as its argument and returns the relevant information in the cmquery data structure.

```c
int cm_bulk_query(int id, cmfdquery *cmquery)
```
An optimisation to allow an application to query multiple flows for information. Takes a macroflow as its argument and return the information in cmfdquery data structure.

```c
int cm_getmflow(int id)
```
Used to get the macroflow corresponding to a stream. Takes the data socket representing the stream as its argument and return macroflow.

```c
int cm_getthresh(int id, cmthresh *cmthresh)
```
 Gets the current value of the threshold parameters for a macroflow (id). These threshold parameters specify when the CM should provide callbacks regarding rate/RTT changes. The threshold parameters are returned in the cmthresh data structure. On failure, -1 is returned. On success, 0 is returned.

```c
int cm_setthresh(int id, cmthresh *cmthresh)
```
Sets the threshold parameters for a macroflow (id). The threshold parameters to be set must be specified in a cmthresh data structure. On success, 0 is returned. Otherwise, -1 is returned.
int cm_close(int id)
   Closes and releases the memory associated with a macroflow. Cleans up the state of the macroflow.

void cm_register_send(void (*handler)(int))
   Registers the function that will be called by the CM whenever it dispatches on a request callback in response to a cm_request() issued earlier by the application. The function is passed the flow for which transmission has been scheduled as its argument. The registered function gets called when the application unblocks on the macroflow due to request callback and issues cm_dispatch() after that.

void cm_register_update(void(*handler)(int, cmquery))
   Registers the function that will be called by the CM whenever it dispatches on a rate callback. The function is passed the flow for which rate callback has happened and the current state of the macroflow in cmquery data structure. The registered function gets called when the application unblocks on the macroflow due to rate callback and issues cm_dispatch() after that.

void cm_dispatch(int fd, int select_read, int select_write, int select_except)
   Used to dispatch an appropriate function based on the type of callback from the CM. The application, which is selecting on the macroflow, after unblocking on the macroflow, normally calls cm_dispatch(). cm_dispatch() then, automatically checks the macroflow for either a rate callback or a request callback and calls the appropriate functions that have been registered for these using cm_register_update() or cm_register_send(). Note that if these functions have not been registered, cm_dispatch() just ignores the relevant callback.
10 WAP TRANSPORT

10.1 Description
Wireless networks, such as GSM, GPRS and UMTS, present a more constrained communication environment compared to wired networks. Because of fundamental limitations of power, available spectrum, and mobility, wireless networks tend to have less bandwidth, more latency, less connection stability, and less predictable availability [WAP00a]. WAP tries to solve the problems introduced by the limitations by specifying an efficient, reliable and secure protocol. In CRUMPET, WAP transport, as specified in FIPA Agent Message Transport Protocol for WAP Specification [FIPA00d], is the protocol to be used over the wireless link, because WAP is, at the time of writing this Deliverable, the only FIPA-specified message transport protocol providing sufficient reliability in the wireless environment. However, as there are many opinions about the goodness of WAP, our design is not limited to WAP. Instead, the design allows a flexible replacement of the wireless transport protocol, if needed.

10.2 Constraints and limitations
The design for WAP transport in CRUMPET depends on the available WAP gateways and clients and the functionality they provide, which is the main constraint for the design. Also, if it is impossible to use mobile Internet operator's (such as Sonera) WAP gateways because of lack of their functionality or security reasons, it must be taken into account that the WAP gateway has to be placed in the same domain with CAN.

The limitations of the Mobile Device pose restrictions for the client side, which has to be as lightweight as possible. Because no prototypes of the MicroFIPA-OS are not implemented and the amount of resources left for nomadic application support functionality, including WAP transport, is unknown, the design for the client side has to flexible in terms of the complexity of the functionality and implementation language used.

10.3 Architecture
Figure 33 depicts the overall WAP architecture to be used in CRUMPET. The whole architecture can be divided into three logical parts: WAP client domain, WAP GW domain and CAN residing in the Internet. In CRUMPET, the CRUMPET client resides in the WAP domain and CAN in the Internet. In between these is the WAP GW domain, which includes both pull- and push WAP gateways. According to WAP specification [WAP01b], pull and push functionality can be implemented in separate gateways or in a single one. In CRUMPET both pull and push functionality are included in a single gateway and is implemented by Kannel WAP gateway [WapIt00a, WapIt00b], because there are no other open-source push-capable WAP gateways available. As soon as there is, one can be used and placed into the mobile operator's domain, because using Kannel in the same domain with CAN and having own dial-in lines will result in scalability problems.

The different domains in the overall WAP architecture are specified in more detail in upcoming subsections.
After the MD is started up and MD sends the first ACL message, a WAP session is created between the MD and ACC on CAN. After the session is created, the ACL message can be sent to the WAP Pull gateway using pull OTA (over-the-air) protocol [WAP00a]. The interaction for this kind of mobile-originated message delivery is shown in Figure 34. The ACL message is delivered along with the MethodInvoke primitive. Whereas in the usual WAP communication the WML page is returned along with the Result primitive, in asynchronous agent communication this primitive only acknowledges that the ACL message has been delivered. After this, the client side WAP stack and its WTP layer take care of acknowledging the received Result primitive.

In the case of mobile-terminated message delivery, the WAP Push [WAP01b] functionality is used. If the session for some reason has not been created or it has died, one is created along with the first message originating from CAN. After that, the ACL message is delivered along with ConfirmedPush primitive. The ACC on MD device acknowledges this by sending back an Ack.
10.4 Interfaces

10.4.1 Mobile device side
Figure 36 shows the elements and interfaces involving WAP messaging on the Mobile Device side.

WAP MTP - WSP interface
WAP MTP, which is a part of the MicroFIPA-OS, is connected to the WAP stack through JNI interface. JNI component represents a c-library, which implements both sending and receiving functionality by calling functions in WSP API. For Kannel FakeWAP client, this API is presented in the following.

Name \texttt{gwlib\_init} - initialise the gw library.

Synopsis \begin{verbatim} void gwlib\_init (void) \end{verbatim}

Description This function initialises the gw library. The FakeWAP client uses the functions in this library. This function must be called as the WAP MTP is brought up.

Parameters \begin{verbatim} void \end{verbatim}

Return value \begin{verbatim} void \end{verbatim}

\footnote{The client's WAP stack includes WSP [WAP00b], WTP [WAP00c] and WDP [WAP00d].}
**gwlib_shutdown** - free gw library resources.

**void gwlib_shutdown (void)**

This function frees all the resources allocated by the gw library. This function should be called once the WAP MTP is shutdown.

**Parameters**

- void

**Return value**

- void

**wap_msg_send**

**static int wap_msg_send (int fd, unsigned char * hdr, int hdr_len, unsigned short tid, int tid_new, unsigned char * data, int data_len)**

**Parameters**

- fd Open file descriptor for the WAP gateway.
- hdr Packet header.
- hdr_len Packet header length.
- tid Transaction ID.
- tid_new New transaction ID.
- data Packet data.
- data_len Packet data length.

**Return value**

- > 0 The number of bytes sent.
- -1 Error

**wap_msg_recv**

**static int wap_msg_recv (int fd, const char * hdr, int hdr_len, unsigned short tid, unsigned char * data, int data_len, int timeout)**

**Parameters**

- fd Open file descriptor for the WAP gateway.
- hdr Packet header.
- hdr_len Packet header length.
- tid Transaction ID.
- data Packet data.
- data_len Packet data length.
- timeout Timeout for waiting for packet.

**Return value**

- > 0 Length of received data.
- == 0 Got an acknowledgement or abort but not the expected data.
- < 0 Error.
The WSP method invokes are implemented as POST-requests. The request comprises of the following (based on FIPA Agent Message Transport Protocol for HTTP Specification [FIPA00e]):

- The URI in the request must be the address of the CRUMPET Access Node’s ACC.
- The request version must be HTTP/1.1
- Headers:
  - `Content-type` should be “x-application/fipa-message”
  - `Host` should be in the form of `hostname:port`
  - `Cache-control` should have the value “no-cache”
  - `MIME-version` should have the value “1.0”
  - `Content-length` must contain the number of bytes in the content of the request.

The request body contains the actual ACL message to be delivered. The message has two components, bit-efficient ACL envelope and bit-efficient ACL message body.

**WDP - CM/UDP interface**

The interface between WDP and UDP as it is in FakeWAP remains unchanged, but the actual UDP implementation of FakeWAP has to be CM-enabled. This implies modification to the `socket.c` module in Kannel. The initialisation function of the socket has to register the socket with CM using the "int cm_open (sockaddr_in dst, sockaddr_in dest)" function. Correspondingly, the shutdown function has to call "void cm_close (int cmid)" function.

A more radical change has to be applied to the sending function, where the structure of the sending loop has to be modified. Before the loop the actual data sending function is registered to CM by calling "cm_register_send (void (*handler) (int))" function, which tells the CM to call the function passed as parameter, when the user requests for sending data. In the loop, as long as there are data left for sending, the CM is asked to send data using "cm_request (int cmid)" function. Eventually, when CM sees it as appropriate, it calls the registered data sending function, which in turn is allowed to send up to MTU bytes of data. This whole process is repeated as long as there are bytes to be sent. Figure 37 depicts this process.

![Figure 37: CM-enabling sending loops.](image-url)
10.4.2 WAP Gateway domain

Figure 38 depicts the WAP gateway architecture in CRUMPET. WAP gateway includes both pull and push gateway functionality\(^{15}\). In CRUMPET, Kannel gateway will be used in implementing this. The only modification, which has to be done to the off-shelf product, is to change the UDP socket functionality as CM enabled as in the client (see section 10.4.1, page 102).

![WAP Pull & Push Gateway](image)

Figure 38: CRUMPET WAP gateway architecture.

10.4.3 CRUMPET Access Node side

![WAP MTP - HTTPReceiver interface](image)

Figure 39: CRUMPET Access Node side WAP architecture.

**WAP MTP - HTTPReceiver interface**

An incoming message from WAP gateway is received at the HTTReceiver. HTTReceiver is able to extract the envelope and ACL message and pass them to the WAP MTP.

**WAP MTP - HTTPSender interface**

For the outgoing message, HTTPSender is used for the purpose. WAP MTP calls the sending method of HTTPSender with bit-efficient envelope and bit-efficient ACL message as parameters. HTTPSender in turn constructs a PAP request [WAP01d], puts the given envelope and ACL message as the content of it and sends it to the WAP Push gateway.

\(^{15}\) The WAP Gateway includes WAE [WAP00e], WSP [WAP00b], WTP [WAP00c] and WDP [WAP00d].
An example of the message format HTTPSender constructs is presented in Figure 40.

```
Content-Type: multipart/related;
boundary=asdlfkjiurwghasf;
type="application/xml"
--asdlfkjiurwghasf
Content-Type: application/xml
<?xml version="1.0"?>
<!DOCTYPE pap PUBLIC "/WAPFORUM//DTD PAP 1.0//EN"
"http://www.wapforum.org/DTD/pap_1.0.dtd">
<pap>
<push-message push-id="9fjeo39jf084@pi.com">
  <address
    address-value="wappush=12345/type=ppg@access.node">
  </address>
</push-message>
</pap>
--asdlfkjiurwghasf
Content-Type: x-application/fipa-message
< Bit-efficient message (including envelope and
ACL message) resides here >
--asdlfkjiurwghasf--
```

Figure 40: Example of PAP request.

10.5 Class diagrams

10.5.1 Mobile Device side

In Figure 41 the Mobile Device side classes are depicted. WAPComms class implements the WAP MTP of FIPA-OS and accesses the WAP stack through a socket. Although socket interface requires that a socket-sender/receiver functionality have to be implemented for the WAP stack, it provides flexibility in terms of changing the WAP stack implementation to another, or to have a whole different wireless transport mechanism for instance while implementing early prototypes (for instance HTTP). Modifications need only to be done inside the socket-sender/receiver to the functionality, which converts the WAP stack's functions to structures that WAPComms understands and sends these structures via the socket.

![Figure 41: Mobile Device side WAP classes.](image-url)
10.5.2 CRUMPET Access Node side

In Figure 42 the Mobile Device side classes are depicted. After the HTTPReceiver has received the message as HTTP request, it extracts the envelope and the ACL message from it and pushes them on to the socket, from which the WAPComms reads them. In the case of sending PAP requests, WAPComms uses HTTPSender class to create the request and to send it. Again as with the Mobile Device side, here we can easily substitute the WAP functionality with HTTP for prototyping, if needed.

![Figure 42: CRUMPET Access Node side WAP classes.](image)

10.6 Class and modules definitions

10.6.1 WAPComms

Package: fipaos.mts.wap
Syntax: public class WAPComms extends ExternalMTPBase
Direct Known Subclasses: none.
All Implemented Interfaces: none.

Design and implementation of WAPComms follows the design guidelines for FIPA-OS [Emo01b].

Constructors:

public **WAPComms** (String name, URL nsUrl)

Parameters:

- name — name of the WAP MTP.
- nsURL — is ignored by the WAPComms.

Methods:

public void **bind** ()

This method is used for binding the MTP to the naming service. In WAP MTP, this method “binds” to the WAP Gateway used.

public void **unbind** ()

This method is used for unbinding the MTP from the naming service. In WAP MTP, this method "unbinds" to the WAP Gateway used.

public Object **lookup** (URL name)

This method is used for looking up the receiving object. In WAP MTP, there is no object, of which method would be called in order to deliver the WAP message.

Parameters: name — URL of the receiver.

Returns: The receiver object.

public Object **lookup** (String name)

This method is used for looking up the receiving object. In WAP MTP, there is no object, of which method would be called in order to deliver the WAP message.

Parameters: name — String-formatted address of the receiver.
Returns: The receiver object.

public void shutdown ()
This method takes care of shutting down the WAP MTP.

public URL getAddress ()
Returns the address of the WAP MTP.

Returns: URL for the WAP MTP.

public Vector getProtocols ()
Returns the protocols supported by the WAP MTP as a vector. String named "wap"
must be the first element of the Vector.

Returns: Vector, of which first element is String "wap".

public void send (Object target, Envelope env, byte[] msg)
Sends the envelope and message to the target. On the CRUMPET Access Node WAP
MTP, this method constructs a PAP request and sends it using HTTPSender. On the
Mobile Device side, the envelope and message are written to a socket, whose peer is
the WAP stack.

Parameters:
target — The receiving object. In the CRUMPET Access Node side WAP
MTP this is HTTPSender and in the Mobile Device side, this is the object
that has the socket connection to the WAP stack.

ev — Envelope to be sent.

msg — ACL message as an array of bytes.

public void handleMessage (WAPMessage msg)
This method receives the WAP messages.

Parameters:
msg — The receiving object. In the CRUMPET Access Node side WAP
MTP this is the HTTPSender and in the Mobile Device side, this is the
object that has the socket connection to the WAP stack.

10.6.2 HTTPSender

Package: fipaos.mts.wap
Syntax: public class HTTPSender
Direct Known Subclasses: none.
All Implemented Interfaces: none.

HTTPSender is a reusable class being able to send out a HTTP-request.

Constructors:
public HTTPSender ()

Methods:
public boolean sendHTTPRequest (Hashtable headers, String payload)
Constructs and sends a HTTP-request.
Parameters:
headers — Hashtable of headers. Each key (String) is the name of the header field and the corresponding value (String) is the value of the header field.

content — String representing the payload of the HTTP-request.

Returns: True, if the HTTP-request was sent successfully. False otherwise.

10.6.3 HTTPReceiver

Package: none.
Syntax: public class HTTPReceiver
Direct Known Subclasses: none.
All Implemented Interfaces: none.

HTTPReceiver listens to HTTP-requests from the WAP gateway. Upon receiving one, HTTPReceiver extracts Envelope and ACL message (as array of bytes) out of the HTTP-request and pass them to WAPComms.

Methods:
public void handleIncoming (byte[] request)
This method receives the HTTP-requests.

Parameters:
request — The incoming request.
11 BIT-EFFICIENT MESSAGING SUPPORT

In wireless environments, it is essential than inter-agent messaging is implemented as efficient as possible. In CRUMPET architecture, the FIPA messaging over wireless link is implemented by using bit-efficient FIPA-ACL [FIPA00c] and bit-efficient envelope [FIPA00b] standard encodings. These concrete encodings minimize the volume of data transferred over the wireless link.

11.1 Description

In bit-efficient FIPA-ACL [FIPA00c] 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 simple string based coding, as it typically reduces additional overhead to half of the original. Furthermore, this improvement is easy implement and generally faster to parse than string based coding—comparing bytes is much faster than comparing strings.

Although the tokenised syntax gives significant improvement compared to string-based representations, true power of bit-efficient FIPA-ACL lies in intelligent caching. 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 tight coupling between communicating peers, and thus it is inapplicable in some situations. Furthermore, implementation of intelligent caching requires more memory and processing power than simple data reduction scheme.

The intelligent caching coding scheme uses the same coding principles and the syntax as the stateless tokenised version. In addition, a code table is used for common tokens such as the strings found from the messages. Because of the code table, a tight coupling between communicating peers is needed. When encoding messages, the encoder must be aware to whom it is sending the message in order to use correct code table. Similarly, the decoder must aware of the identity of the message sender.

![Figure 43: Using the code tables between two communicating ACCs](image)

Figure 43 depicts the code table usage between two communicating ACCs. Both peers have two code tables, one for sending messages and one for receiving. When a sending ACC (S) encodes a message, it looks for from its encoder code table (e'), whether component to be coded can be found from there. If so, the cache reference is encoded the message. Otherwise, the component is inserted to encoder code table, and is sent to receiver as it is. When the receiver ACC (R) receives a message, it knows when parsing the message whether the component is found from its decoder code table (d') or not. If it is not found, the new component is inserted to code table. Now, when the ACC R sends some other message to ACC S, it uses its encoder code table (e'') to encode the message, and ACC S uses decoder code table (d'') in order to decode the message. Especially, the sender is never allowed to use the decoder code table when sending messages. Using both decoder and encoder code tables for encoding messages, more efficient coding is possible, but the code table management is significantly harder. In scheme, there is no need for explicit synchronisation messages but both ends can easily maintain their code tables.
Figure 44: Bit-efficient FIPA-message translation at the CRUMPET Access Node.

Figure 44 depicts the usage of bit-efficient FIPA-messaging in the CRUMPET architecture. The bit-efficient FIPA-ACL and FIPA-envelope are used between the mobile terminal and the CRUMPET Access Node. The Transport Interoperability Service (TIS) at the CRUMPET Access Node converts the mobile originated bit-efficiently encoded messages to some concrete representation that is used in communication in fixed network, and mobile terminated messages encoded using some concrete representation to bit-efficient format. This conversion happens transparently to communicating agents. If the message is destined to an agent situated at the CRUMPET Access Node, no conversion is made.

11.2 Constraints and Limitations

In order to make the usage of bit-efficient FIPA-ACL easy, but without losing any of its benefits, fixed code table size is used. The code table size to be used is read from platform profile when starting the agent platform. Note that code tables are used only in FIPA-ACL encoding. In bit-efficient FIPA-envelope encoding code tables are not used.

11.3 Interfaces

ACLOutputStream reads in FIPA-OS Message objects and outputs a sequence of bytes. ACLInputStream in turn reads in a sequence of bytes and provides user with FIPA-OS Message object. Therefore, bit-efficient encoder/decoder functionality is implemented in FIPA-OS ParserService, which implements both PreParser and PostParser Services. ParserService selects the appropriate parse/deparse mechanism based upon the ACLRepresentation field of the Envelope.

ParserService usually resides on the external service stack of ACC, but if no ACC will be used in the MicroFIPA-OS, the ParserService is added to the internal service stack of the MicroFIPA-OS agent(s)'s internal service stack.
11.4 Class diagrams

In the Figure 46 the classes composing the bit-efficient FIPA-ACL encoder/decoder functionality are depicted. The classes visible for the users of BE are ACLOutputStream and ACLInputStream. The former takes care of writing a given FIPA-OS Message bit-efficiently to an output stream. Respectively, the latter reads in a sequence of bytes representing the bit-efficient FIPA-ACL message and returns a FIPA-OS Message object. If the incoming message from fixed network is already bit-efficient encoded, the BEService does nothing to it.

ACLOutputStream delegates the actual encoding functionality to the ACLEncoder class. ACLEncoder in turn uses EncoderCodeTable class, if dynamic codetables are used. BinDate and BinNumber classes, which are utilized by both ACLEncoder and ACLInputStream, are specialized in encoding/decoding dates and number, respectively. ACLInputStream uses DecoderCodeTable class, if dynamic codetables are enabled. ACLConstants class includes all the constant variables.
Figure 46: BE-ACL class diagram.

In the Figure 47 the classes composing the bit-efficient FIPA-Envelope encoder/decoder functionality are depicted. The classes visible for the users of BE are EnvelopeEncoder and EnvelopeDecoder. The former takes care of constructing a given FIPA-OS Envelope bit-efficiently. Respectively, the latter reads in a sequence of bytes representing the bit-efficient FIPA-Envelope and returns a FIPA-OS Envelope object.

Figure 47: BE-Envelope class diagram

11.5 Class definitions

11.5.1 ACLConstants

Package: fipaos.parser.acl.bitefficient

Syntax: public interface ACLConstants

All Known Implementing Classes: ACLEncoder, ACLInputStream, ACLPerformatives, BinDate

ACLConstants interface defines fipa-bitefficient-std ACL message transport syntax related constants.
11.5.2 ACLEncoder

Package: fipaos.parser.acl.bitefficient
Syntax: public class ACLEncoder
Direct Known Subclasses: None.
All Implemented Interfaces: ACLConstants

ACLEncoder implements an encoder for bit-efficient ACLMessages.

Constructors:
public ACLEncoder()
 Initialise ACL encoder with no codetable coding scheme.

d public ACLEncoder(int size)
 Initialise ACL encoder with given codetable size.

Parameters: size — code table size in bits (between 8 and 16)

d public ACLEncoder(EncoderCodetable ct)
 Initialise ACL encoder with given code table.

Parameters: ct — Codetable to be used in encoding process.

Methods:
public byte[] encode(ACLMessage msg)
 Encodes an ACL message.

Parameters: msg — Message to be encoded

Returns: Byte array containing the coded message.

d public EncoderCodetable getCodetable()
 Returns the codetable associated with this encoder

Returns: Codetable associated with this encoder

11.5.3 ACLInputStream

Package: fipaos.parser.acl.bitefficient
Syntax: public class ACLInputStream
Direct Known Subclasses: None.
All Implemented Interfaces: ACLConstants

ACLInputStream reads fipa-bitefficient-std coded ACL messages from given InputStream.

Constructors:
public ACLInputStream(java.io.InputStream i)
 Initialise the ACL input stream with given InputStream

Parameters: i — InputStream from which messages are read.

d public ACLInputStream(java.io.InputStream i, int size)
 Initialise the ACL input stream with given InputStream and codetable size

Parameters:
 i — InputStream from which messages are read
 size — size for the codetable
public ACLInputStream(java.io.InputStream i, EncoderCodetable ct)
Initialise the ACL input stream with given InputStream and codetable

Parameters:
  i — InputStream from which messages are read
  ct — codetable to be in encoding process.

Methods:
public ACLMessage readMsg()
Reads an ACL message from the input stream

Returns: The ACL message read.

11.5.4 ACLOutputStream

Package: fipaos.parser.acl.bitefficient
Syntax: public class ACLOutputStream
Direct Known Subclasses: None.
All Implemented Interfaces: Node

ACL OutputStream that writes fipa-bitefficient-std coded ACL messages into a stream.

Constructors:
public ACLOutputStream(java.io.OutputStream o)
Initialise the ACL output stream with given OutputStream

Parameters: o — OutputStream to which message are written.

public ACLOutputStream(java.io.OutputStream o, int size)
Initialise the ACL output stream with given OutputStream and codetable size

Parameters:
  o — OutputStream to which message are written.
  size — Size for the codetable.

Methods:
public void write(ACLMessage msg)
Writes an ACL bit-efficient message to the OutputStream.

Parameters: msg — ACLMessage to be written.

11.5.5 BinDate

Package: fipaos.parser.acl.bitefficient
Syntax: public class BinDate
Direct Known Subclasses: None.
All Implemented Interfaces: ACLConstants

BinDate implements Bitefficient representation of “DateTimeToken”.

Methods:
public String fromBin(byte [] b)
Converts bit-efficient Date to String.

Parameters: b — Bit-efficient date

Returns: String containing ASCII representation of supplied date.
public byte[] toBin(String s)
    Converts ASCII representation of Date to bit-efficient representation.

    Parameters: s — String containing ASCII representation of date

    Returns: Bit-efficient representation of supplied date.

11.5.6  BinNumber

Package: fipaos.parser.acl.bitefficient
Syntax: public class BinNumber
Direct Known Subclasses: None.
All Implemented Interfaces: None.

BinNumber implements bit-efficient number representation.

Methods:
public String fromBin(byte [] b)
    Converts bit-efficient number representation to String.

    Parameters: b — byte array containing bit-efficient representation of a number

    Returns: String containing ASCII representation of a supplied number.

byte[] toBin(String s)
    Converts ASCII representation of number to bit-efficient representation.

    Parameters: s — number in ASCII representation

    Returns: Byte array containing bit-efficient representation of supplied number.

11.5.7  BinRep

Package: fipaos.parser.acl.bitefficient
Syntax: public class BinRep
Direct Known Subclasses: BinDate, BinNumber.
All Implemented Interfaces: None.

BinRep implements generic conversion routines between ASCII and bit-efficient representation of numbers. Numbers are coded by reserving four bits for each digit in the number’s ASCII representation, that is, two ASCII numbers are coded into one byte. Table 13 shows a 4-bit code for each number and special codes that may appear in ASCII coded numbers.

If the ASCII presentation of a number contains an odd number of characters, the last four bits of the coded number are set to zero (the Padding token), otherwise an additional 0x00 byte is added to the end of the coded number. If the number to be coded is an integer, decimal number, or octal number, the identifier byte 0x12 is used. For hexadecimal numbers, the identifier byte 0x13 is used. Hexadecimal numbers are converted to integers before coding (the coding scheme does not allow characters from a through f to appear in number form).
Table 13: Binary representation of number tokens

<table>
<thead>
<tr>
<th>Token</th>
<th>Code</th>
<th>Token</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Padding</td>
<td>0000</td>
<td>7</td>
<td>1000</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>8</td>
<td>1001</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>9</td>
<td>1010</td>
</tr>
<tr>
<td>2</td>
<td>0011</td>
<td>+</td>
<td>1100</td>
</tr>
<tr>
<td>3</td>
<td>0100</td>
<td>E</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>0101</td>
<td>-</td>
<td>1110</td>
</tr>
<tr>
<td>5</td>
<td>0110</td>
<td>.</td>
<td>1111</td>
</tr>
<tr>
<td>6</td>
<td>0111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Methods:

protected static byte decode(int i)
Converts bit-efficient number to ASCII character.

Parameters: i — bit-efficiently coded number

Returns: ASCII representation of supplied number.

protected static byte encode(int i)
Converts one ASCII character (number) to bit-efficient representation.

Parameters: i — ASCII character representing the number to be coded

Returns: Bit-efficiently encoded representation of supplied number.

11.5.8 DecoderCodetable

Package: fipaos.parser.acl.bitefficient
Syntax: public class DecoderCodetable
Direct Known Subclasses: None.
All Implemented Interfaces: None.

Implementation of decoder codetable for fipa-bitefficient-std messages.

Constructors:

public DecoderCodetable(int size)
Initialise the codetable with given size.

Parameters: size — Size of the codetable in bits (between 8 and 16)

Methods:

public String lookup(int code)
Returns the String to which the specified code is mapped in this codetable. The entry in the codetable is moved to the end of LRU list.

Parameters: code — The code to lookup

Returns: The String to which the code is mapped in this codetable; null if the code is not mapped to any String in this hashtable.
public int insert(String s)
    Insert a new String to codetable. If the code table is full, \( (\text{size} \gg 3) \) entries are removed from the end of LRU list.

    Parameters: \( s \) — String to insert

    Returns: Index of added String.

11.5.9 EncoderCodetable

Package: fipaos.parser.acl.bitefficient
Syntax: public class EncoderCodetable
Direct Known Subclasses: None.
All Implemented Interfaces: None.

EncoderCodetable implements the encoder codetable for fipa-bitefficient-std.

Constructors:
    public EncoderCodetable(int size)
    Initialise the codetable with given size.

    Parameters: size — Size of the codetable in bits (between 8 and 16)

Methods:
    public String lookup(int code)
    Returns the String to which the specified code is mapped in this codetable. The entry in the codetable is moved to the end of LRU list.

    Parameters: code — The code to lookup

    Returns: The index of the string if found, -1 otherwise

    public int insert(String s)
    Inserts a string to codetable. If the string is already in code table, its code is returned, and the string is moved to the end of LRU list.

    Parameters: \( s \) — String to insert

    Returns: Index of added String.

11.5.10 EnvelopeConstants

Package: sonera.fipa.envelope
Syntax: public interface EnvelopeConstants
All Known Implementing Classes: EnvelopeEncoder, EnvelopeDecoder

EnvelopeConstants interface defines bit-efficient envelope transport syntax related constants.

Constructors:
    None

Methods:
    None
11.5.11 EnvelopeEncoder

Package: sonera.fipa.envelope
Syntax: public class EnvelopeEncoder
Direct Known Subclasses: None.
All Implemented Interfaces: EnvelopeConstants

EnvelopeEncoder implements an encoder for bit-efficient envelopes.

Constructors:
  public EnvelopeEncoder()
  Initialise the envelope encoder.

Methods:
  public byte [] encode(Envelope e)
  Translates an envelope to bit-efficient representation.

  Parameters: e — Envelope to be encoded
  Returns: Byte array containing bit-efficient representation of supplied Envelope

11.5.12 EnvelopeDecoder

Package: sonera.fipa.envelope
Syntax: public class EnvelopeDecoder
Direct Known Subclasses: None.
All Implemented Interfaces: EnvelopeConstants

EnvelopeDecoder implements a decoder for bit-efficient envelopes.

Constructors:
  public EnvelopeDecoder()
  Initialise the envelope decoder.

Methods:
  public Envelope getEnvelope(byte [] data)
  Translates a bit-efficiently encoded envelope to platform internal representation.

  Parameters: b — byte array containing bit-efficiently encoded envelope
  Returns: Decoded envelope

11.6 Bit-Efficient Content Language

Since no commitment to any content language has yet been done, bit-efficient representation of such content language remains an open issue. As soon as CRUMPET commits to specific content language, a separate document will be published containing documentation how selected content language will be encoded when transferring messages over a wireless link.
12 DISCONNECTIONS AND BUFFERING

12.1 Description
The Message Buffering Service (MBS) provides services for buffering messages when particular agent/AP cannot be reached\(^{16}\). Agents does not have to be aware of MBS, but the underlying agent platform can take care of the details in order to enable buffering as well as requesting message forwarding.

Figure 48 depicts the general buffering architecture in CRUMPET.

![Buffering architecture in CRUMPET](image)

The Mobile Device's MicroFIPA-OS and its MTS include a MessageBufferService, which is composed of the actual Buffer and BufferRouter components. The MicroFIPA-OS Buffer, which contains ACL messages objects (together with Envelopes) with various transport addresses, acts as an "outbox" buffer for the messages that are delivered to the CRUMPET Access Node, whereas the Buffer in the CRUMPET Access Node FIPA-OS acts as an "inbox" buffer for the messages that are delivered to the Mobile Device. The local incoming messages are not routed via the Buffer. MTPs have their own buffers, which will be filled by messages from the MessageBufferService's Buffer fed by the BufferRouter. When BufferRouter feeds message(s) to the MTP for transmission, it assigns a sequence number to each message. It also keeps the original message in the MessageBufferService's Buffer and therefore passed a copy of the original message for transmission. As soon as MTP has delivered the

\(^{16}\) This information is provided by Monitor Agent.
message successfully, it informs the BufferRouter by passing the sequence number of the delivered message back to the BufferRouter. At this point the original message is removed from the MessageBufferService's Buffer. An original message is removed also from the MessageBufferService's Buffer when the MTP informs that the message could not be delivered. In this case also the sending agent is informed with an error.

12.2 Buffering parameters
Buffering time for a message can be controlled with two parameters: deliver-within and deliver-after. Deliver-within implies that the buffer should deliver the message within the specified time constraint. If the message is not delivered within the specified time, the message is destroyed from the buffer and a failure is sent to the sending agent. Deliver-after parameter can be used to tell the buffer that the message is not allowed to be transmitted before the deliver-after time has elapsed. Agent may set these parameters for a message using the buffering-parameter in message envelope. An example of a message envelope containing parameters for buffering is presented in Figure 49. Although the example uses XML-encoding for the envelope, the same scheme applies to the bit-efficient envelope encoding.

```xml
<?xml version="1.0"?>
<envelope>
  <params index="1">
    <to>
      <agent-identifier>
        <name>receiver@foo.com</name>
        <addresses>
          <url>wap://foo.com/acc</url>
        </addresses>
      </agent-identifier>
    </to>
    <from>
      <agent-identifier>
        <name>sender@bar.com</name>
        <addresses>
          <url>wap://bar.com/acc</url>
        </addresses>
      </agent-identifier>
    </from>
    <acl-representation>fipa.acl.rep.xml.std</acl-representation>
    <date>20000508T042651481</date>
    <encrypted>no encryption</encrypted>
    <buffering-parameters>
      <deliver-within>60</deliver-within>
      <deliver-after>5</deliver-after>
    </buffering-parameters>
  </params>
</envelope>
```

Figure 49: Buffering parameters in message envelope.

12.3 Mobile-terminated message buffering
The Mobile Device's inbound buffer resides on the “safe” side, i.e. on the CRUMPET Access Node FIPA-OS. When a message from fixed network (for instance from some CRUMPET service agent) arrives at the CRUMPET Access Node FIPA-OS's ACC, it is delivered to the ACCRoutingService. ACCRoutingService is able, by looking at the destination address, to decide, if the message is destined to some local agent registered to the CRUMPET Access Node FIPA-OS or if it should be forwarded to the Mobile Device. In this case we assume that the message is destined to the Mobile Device, which means that the message is routed to the MessageBufferService. Upon arriving at the MessageBufferService, the message is put to the buffer. BufferRouter is responsible to feed messages to the WAP MTP to keep it busy all the time in sending messages. BufferRouter creates copies of the messages, marks them as delivered and passes the set of message copies to the WAP MTP. WAP MTP in turn sends out the messages using HTTPSender, which creates PAP requests out of the messages and
sends them to the WAP Gateway. As soon as the WAP Gateway receives an acknowledgement indicating that a given message was delivered successfully, it informs the WAP MTP. In this case, the WAP MTP informs the BufferRouter by telling that the given message was delivered successfully and that the original message can be destroyed from the MessageBufferService's Buffer. As soon as the WAP MTP "acks" some message as successfully delivered, BufferRouter feeds new message(s) to the WAP MTP. It can feed one message or a set of messages depending on the ability of WAP MTP to handle the amount of new messages or on the line rate of the wireless link.

If the wireless connection between the WAP Gateway and the Mobile Device is down, the WAP Gateway first tries to send the messages as long as its internal sending timer expires. In this case the WAP MTP is informed about the failure in sending the message. If the message's deliver-within time has not expired, WAP MTP tries to resend the message as long as the buffering timer expires. Should this happen, the BufferRouter is informed about the sending failure. The BufferingRouter destroys the original message from the MessageBufferService's Buffer and informs the sending agent with an error.

12.4 Mobile-originated message buffering
The Mobile Device's outbound buffer resides on the MicroFIPA-OS. When an agent on the Mobile Device sends a message, it is routed to the MessageBufferService, which puts the message in to the MessageBufferService's Buffer. As in the inbound message buffering, the BufferRouter feeds message copies to MTPs from the MessageBufferService's Buffer. MTPs in turn send messages at their own pace and inform BufferRouter as soon as message was delivered successfully so that BufferRouter can destroy the original message from the MessageBufferService's Buffer.

12.5 Constraints and Limitations
None.

12.6 Interfaces
MessageBufferService is implemented as FIPA-OS service. Thus it must implement two methods: incoming () and outgoing (). The former is meant to pass an incoming message to the MessageBufferService. This is called by the MTPs. The latter is meant for passing outgoing messages to the MessageBufferService to be buffered. This method is called by the service residing on top of MessageBufferService. In addition to these, MessageBufferService provides MTPs with an interface for asking copies of the buffered messages for sending and informing about the success or failure of the message sending. For detailed design of the classes see the upcoming sections 12.7 and 12.8.

12.7 Class diagram
Figure 50 depicts the classes involved with the buffering. MessageBufferService is a FIPA-OS specific class, which implements the FIPA-OS PreParserService interface. Platform-independent classes, BufferRouter and Buffer, implement the actual functionality of the buffering. When an outgoing message arrives at the MessageBufferService's outgoing-method, the buffering of the message is delegated to the BufferRouter. Incoming messages arriving at the MessageBufferService's incoming-method require no special handling. Thus, they are forwarded directly to the upper service layer.
12.8 Class definitions

12.8.1 MessageBufferService

Package: fipaos.mts.service
Syntax: public class MessageBufferService implements PreParserService
Direct Known Subclasses: None.
All Implemented Interfaces: PreParserService.

MessageBufferService is a FIPA-OS service implementing the PreParserService interface.

Methods:

public void **outgoing** (Message msg)
This method is called by the upper service layer and is meant to buffer an outgoing message. The actual buffering is delegated to the BufferRouter by calling its addToBuffer method.

Parameters: msg — message containing both ACL message and envelope.

public void **incoming** (Message msg)
This method receives an incoming message from MTPs. This method does nothing else but forwards the message directly to the upper service layer.

Parameters: msg — message containing both ACL message and envelope.

public void **shutdown** ()
This method shuts down the MessageBufferService. It also propagates the shutdown request to the BufferRouter.

public void **initialise** (PreParserService upper, PreParserService lower)
This method initialises the MessageBufferService.

Parameters:
upper — the upper service layer to which the incoming messages are to be forwarded.

lower — the lower level service layer. In this case the lower layer is the MTPs, so this is set to null.

### 12.8.2 BufferRouter

*Package:* fipaos.mts.service  
*Syntax:* public class BufferRouter  
*Direct Known Subclasses:* None.  
*All Implemented Interfaces:* None.

BufferRouter implements the buffering functionality. It receives outgoing messages from the MessageBufferService and buffers them. Messages are buffered based on the MTP so that for every MTP there is a message buffer. A message can reside only in one buffer. If a message has more than one destination address, the initial buffer will be selected by the first address. In the case, where some MTP is not able to send the message, and it is desired to try with some other MTP, the message must be moved to the new buffer.

**Methods:**

- **public void addToBuffer (Message msg, String buffer)**  
  This method stores a message to a buffer based on the type parameter. It must be noted that the destination address list of the message must include at least one proper address, i.e. a valid address for the MTP, to which the buffer in question is associated with.

  **Parameters:**  
  - msg — message containing both ACL message and envelope.  
  - buffer — the buffer to which the message is to be added.

- **public Vector getMessages (String buffer, int count)**  
  This method fetches copies from the specified buffer.

  **Parameters:**  
  - buffer — the buffer from which the messages are fetched.  
  - count — the number of message copies to be fetched.

- **public void sentSuccessfully (Message msg)**  
  This method is called by MTP upon successful message sending. This method removes the original message from the appropriate buffer.

  **Parameters:**  
  - msg — message containing both ACL message and envelope.

- **public void sendingFailed (Message msg)**  
  This method is called by MTP upon a failure in message sending. Based on the configuration of the BufferRouter, this method either removes the original message from the appropriate buffer and sends a failure notification to the sender agent, or it tries to send the message using some other MTP by moving the original message to the other buffer.

  **Parameters:**  
  - msg — message containing both ACL message and envelope.

- **public void shutdown ()**  
  This method shuts down the BufferRouter by clearing the buffers.
12.8.3 Buffer

Package: fipaos.mts.service
Syntax: public class Buffer
Direct Known Subclasses: None.
All Implemented Interfaces: None.

Buffer stores a vector of messages for a given MTP.

Methods:
    public Vector getMessages (int count)
    This method fetches copies from the buffer.

Parameters: count — the number of message copies to be fetched.
13 REFERENCES


