

AGORA

*An integrated approach for
collaboration in MANETs*

by

Marcel Arrufat Arias

*Security and Computer Engineering
Master's Degree*

University Rovira i Virgili (Tarragona)

*Maths and Computer Engineering
Department*

Advisor: Pedro García López

May 2008

Overview

1.	INTRODUCTION.....	3
1.1.	BACKGROUND: MOBILE AD-HOC NETWORKS AND COLLABORATIVE ENVIRONMENTS	5
2.	ANALYSIS AND RELATED WORK	7
2.1.	SYSTEM REQUIREMENTS	8
2.1.1.	<i>Plug-in framework requirements</i>	<i>8</i>
2.1.2.	<i>Middleware requirements.....</i>	<i>8</i>
2.1.3.	<i>Network requirements</i>	<i>9</i>
2.2.	STATE OF THE ART	10
2.2.1.	<i>Middleware approaches</i>	<i>10</i>
2.2.1.1.	<i>Communication toolkit: JGroups</i>	<i>12</i>
2.2.2.	<i>Multihop networking</i>	<i>14</i>
2.2.2.1.	<i>Multihop unicast Routing.....</i>	<i>14</i>
2.2.2.2.	<i>Application Level Multicast</i>	<i>14</i>
2.3.	CONCLUSION	17
3.	ARCHITECTURE AND DESIGN.....	18
3.1.	OVERALL DESIGN	19
3.2.	NETWORK LAYER	20
3.2.1.	<i>Functional Description.....</i>	<i>20</i>
3.2.2.	<i>Architecture.....</i>	<i>21</i>
3.3.	MIDDLEWARE LAYER	24
3.3.1.	<i>Functional description</i>	<i>24</i>
3.3.2.	<i>Architecture.....</i>	<i>27</i>
3.4.	PLUG-IN FRAMEWORK	36
3.4.1.	<i>Functional Description.....</i>	<i>36</i>
3.4.2.	<i>Architecture.....</i>	<i>37</i>
4.	DEVELOPMENT	40
5.	EVALUATION	42
6.	CONCLUSION	47
7.	REFERENCES.....	48
8.	GLOSSARY	49
	ANNEX 1. RELATED PUBLICATIONS	50

1. Introduction

The objective of this project is to build a complete platform that allows spontaneous collaboration in mobile ad-hoc networks (MANETs) in an easy and flexible way. However, developing applications specially targeted for MANETs is not a trivial task. Devices' limited resources together with dynamic and multihop network present a serious challenge which applications must face. In these terms, it seems reasonable that middleware for ad hoc networks will highly help in reducing the complexity of MANET application development. Middleware approaches provide high level services which application can use in order to construct more complex and flexible applications.

It is known that, due to MANET characteristics, there is not a unique middleware solution that copes with all needed requirements. Several and different challenges arise when facing these requirements. In first place, efficient use of resources, such as memory, bandwidth and computational power, is needed. System scalability becomes crucial when a great amount of members join the network and try to intercommunicate. Other issues like quality of service, devices' heterogeneity and security may also be considered when creating a middleware for ad hoc networks. Therefore, it seems that requirements may vary depending on the selected scenario for ad hoc networks.

Regarding mobile communication, during the last few years the reduction of the cost of portable devices has implied a growing utilization of mobile phones, handheld game consoles and pocket computers. In consequence, a new range of opportunities arise for collaborative working environments. However, bringing the features of collaborative systems to the mobile ad-hoc (MANET) scenario is not trivial. Although flexibility and low cost establishment make these networks attractive for spontaneous collaboration, several management and communication problems emerge when traditional collaboration systems are moved towards the MANET environment. Topology awareness, node dynamicity, lack of central servers and scarce resources are new elements that change the traditional rules in collaborative working environments.

Communication functionalities stand as one of the most important constraints since one-to-one and group communication are the key on which collaborative applications rely. Collaborative applications frequently need chat rooms, file sharing and e-mail messaging so synchronous and asynchronous message delivery seems necessary. Since using TCP as transport protocol seems to be ineffective in MANETs, lighter approaches using UDP seem to be the most suitable solution. However, these applications may need a reliable communication channel in order to ensure high packet delivery. Besides, although it is usually not mentioned, an ordered channel is also necessary for delivering packets for most applications. Even though these assumptions may be too costly for large groups, this project's scope is restricted to small, medium-sized groups, so providing these needs is feasible. We should not forget that

communication efficiency plays a fundamental role in MANET communication due to the multihop nature of the network. Achieving efficient group communication is one of the foundations of collaborative applications. On the other hand, group membership and management are also necessary in developing group-aware collaborative application. Notification of online and joining/leaving members is useful for most collaboration systems, which usually use membership information to perform collective activities.

Regarding issues other than collaboration requirements, middleware approaches should be easy to deploy, extend and use. Usability and ease of application development turn middleware in a powerful tool to develop end-user applications. Current approaches tend to offer a restricted set of functionalities: publish a message under a certain topic, share a file or retrieve information by using pattern-matching. These functionalities are in fact useful but may not be sufficient in order to develop more complex MANET applications in a fast and straightforward manner.

Taking all these considerations into account, the main directions we will take under the development of this project are the following:

- The construction of a collaboration middleware provided with a full set of communication mechanisms and membership information.
- The development of a plug-in framework which benefits from middleware services in order to build final applications in a rapid and simple manner.
- The utilization of an application level multicast (ALM) protocol designed to enhance communication performance, which avoids using a specific MANET multicast.

With all these components, we state that AGORA offers a new ready-to-deploy solution for developing collaborative applications for MANETs in an easy and straightforward way. Since it is not restricted to offering just a single service like file-sharing or probabilistic multicast delivery, developers can use a full set of functionalities which fit the needs of each application.

As we will see hereafter, the collaboration middleware provides synchronous and reliable communication: communication channels, naming and publish/subscribe services. Besides, the topology-aware multicast protocol takes care of minimizing global communication, while the plug-in framework is in charge of reducing application development complexity.

1.1. BACKGROUND: MOBILE AD-HOC NETWORKS AND COLLABORATIVE ENVIRONMENTS

Mobile Ad Hoc networks (MANETs) are formed by heterogeneous devices that communicate without any existing infrastructure. Ease of deployment and decentralized administration have turned MANETs in an attractive scenario for building different kind of applications like management in emergency/disaster situations, battlefield coordination and many others. Among these, collaborative working environments (CWE) face a promising future due to proliferation of handheld and mobile devices together with the need of overcoming the limitations of current collaboration schemes. Decentralization, self-adjusting necessities and dynamic behavior must meet in order to achieve efficient collaboration mechanisms in the MANET environment.

On one hand, we must consider that mobile ad-hoc networks are tied to several restrictions. The principal is due to the multi-hop concept. This means that each node will act as a sender/receiver but also as a packet router. Nodes may be needed to do so, since transmission range is limited, and two nodes located far away from each other may have on common neighbor that acts as a “bridge” between them. Furthermore, this topology, or form of the network, may vary with the movement of the mobile nodes. The other concept apart from the mobile one is the ad-hoc nature of the network. MANETs do not need any previous infrastructure, i.e. access points, wired communications, etc. Nodes are just ready to communicate by having a wireless interface, as long as they stay in wireless range.

On the other side, collaboration scenarios are characterized by having a moderate number of nodes located in a small area. Normally, they are located in a room or in a large hall, where some of them may remain static for long periods of time. Some nodes may change its location from time to time in order to interact with other existing groups. However, most communication is performed in well-defined areas, where nodes are located at most two or three hops from the most distant node. Besides, most scenarios and especially collaborative working environments depend highly on group communication. One-to-one and one-to-many communication primitives are essential for the development of these applications. In the past years, several alternatives for group communication in MANETs have been studied, such as broadcast, multicast, and even geocast. From these candidates, multicast seems the most suitable solution since it makes no assumptions about devices’ location and, at the same time, is more efficient than message broadcasting for most scenarios.

There are two existing approaches for multicast message delivery. In first place, most of the proposed multicast protocols like MAODV or ODMRP follow network layer approaches. These protocols tackle bandwidth, quality of service and other issues for multicast communication. However, none of these protocols have mature and widely tested implementations. The second alternative for enabling multicast communication is application

layer multicast (overlay multicast) in which multicast packets are encapsulated in unicast datagrams and delivered to all group members. In application multicast only group members need to keep state information. Moreover, application layer multicast provides ease of deployment as well as capability of hiding underlying link errors. Hence, an overlay multicast protocol could be easily adapted to collaborative applications requirements.

2. Analysis and related work

Previous to the design and implementation process, we have identified the services that each of the main components of the system should offer. We will explain briefly and evaluate current features from existing MANET middleware approaches: group management, membership and other communication paradigms, together with application level multicast for ad-hoc networks.

Once these requirements have been analyzed, we have studied which of these needs are already solved by the existing approaches found in the literature.

2.1. SYSTEM REQUIREMENTS

The following paragraphs detail all system requirements. We must consider that not all requirements are exactly as difficult to accomplish and some of them may be grouped into more complex requirements. As explained in the introduction, we will divide the development in three different layers: the plug-in framework, the middleware layer and the network layer.

Whereas numerous network layer protocols can be found in the literature [1], not as many overlay multicast protocols have been considered. In the following section we will introduce some of the most important application level multicast approaches.

2.1.1. PLUG-IN FRAMEWORK REQUIREMENTS

The plug-in framework must benefit from middleware services in order to build final applications in a rapid and simple manner. The plug-in framework represents the main point of interaction between the user and our system, so all graphical user interface features, as well as the final applications will be accessible from the framework.

Applications launched in the framework must be encapsulated and should be easy to develop, deploy and run. These applications will be developed as plug-ins so the framework will be in charge of the following functionalities:

- Installing new plug-ins in run-time.
- Retrieving plug-ins from other hosts in the network.
- Controlling plug-in lifecycle: Start and stop plug-ins.
- Simple packaging for plug-ins, allowing information and resources to be attached to the plug-in in a single file.
- As an additional feature, allowing plug-in start in run-time after downloading from other host.

2.1.2. MIDDLEWARE REQUIREMENTS

The middleware, which is the core of the whole system, needs a rich variety of functions in order to allow easy development of collaborative applications:

- Group awareness: each node will be identified as a member inside a group. All actions, such as communication, will be performed inside the scope of a group.
- Membership: a list of current members connected to the group is necessary for middleware services and plug-ins to build more complex services and applications.

- One-to-one communication: unicast communication to allow direct messaging between two members.
- Group communication: multicast communication to allow sending messages to all group members.
- Publish/Subscribe channels: asynchronous messages sent under a certain topic must be available for applications.
- Naming services: name resolution, service and resource discovery are really necessary for building distributed applications.

2.1.3. NETWORK REQUIREMENTS

The network layer is the bottom-most layer and on which all upper layers rely. Building efficient primitives on this layer is fundamental for showing good performance on the rest of layers. The network components should provide two different but both important characteristics. On the one hand, it is clear that they must offer basic communication primitives to the middleware layer, and at the same time should minimize the impact of message routing in the multihop network. Since most multicast protocols are bound to a specific operative system due to creation of cross-layer communication, it would be interesting to remove this constraint by using an application level multicast for multicast delivery.

To summarize, we believe these are the main necessities in this layer:

- Remove the constraint of using a network-bound multicast protocol.
- Enable efficient application multicast delivery by considering the specific needs of a collaborative scenario.
- Offer basic communication primitives to middleware services.

2.2. STATE OF THE ART

Previous to the design phase, we have studied the existing different proposals in the literature. On the one hand, we have evaluated different middleware proposals in order to use a substrate which all the services can be built on. On the other hand, several application multicast protocols have been examined in order to maximize performance on message delivery. Since application level multicast relies on an underlying unicast protocol, we will explain briefly the possible candidates for performing unicast routing. However, we will not make a further study in available plug-in frameworks, since the framework stands as a proof-of-concept of the middleware. Furthermore, we aim to develop some specific characteristics which may difficultly fit into existing approaches.

2.2.1. MIDDLEWARE APPROACHES

In first place, evaluated middleware must offer services in different areas, such as group management or basic communication which will provide the foundations to build higher layer functionalities for plug-ins and plug-in management. Besides, due to MANET characteristics, mobile and multi-hop scenario must be supported. Also, scalability and small traffic overhead are key features that must be included to provide an efficient communication platform. In terms of software maturity, stable and widely tested implementations have to be considered, as well as the portability of the solutions (e.g. implementations not tied to a specific routing protocol or platform).

It is difficult to establish an obvious taxonomy of available middleware solutions due to the tightly coupling of middleware with specific applications. However, the following classification is proposed based on the programming model. Hereafter we will review several middleware approaches for MANETs that share common features and tackle similar challenges than our proposal. Following the mentioned classification, the reviewed solutions could be classified as Peer to Peer Based Middleware (JMobiPeer, Peer2Me) and Event Based and Message Oriented Middleware (STEAM, EMMA, AGAPE). Event Based Middleware is used to support distributed applications that must react to changes in the environment, which is very suitable for MANETs because of their lack of infrastructure. Likewise, Message Oriented Middleware provides asynchronous communication paradigms like publish/subscribe which are particularly adequate for pervasive environments.

One of these approaches is STEAM [2], an event-based middleware that eliminates dedicated event servers and instead uses the implicit publish/subscribe model. Consumers subscribe to certain event types and publishers are able to publish particular events. Moreover, STEAM allows different filters to be applied to the published events. Content filters are used to define complex subscriptions at the subscriber's side. Likewise, proximity filters can be defined in the publisher's side in order to restrict the propagation of such events.

In EMMA [3], a well known standard from traditional distributed systems is adapted to cope with MANET requirements. EMMA is an implementation of the Java Message Service (JMS) that incorporates an epidemic routing mechanism to facilitate message delivery. This middleware provides point-to-point communication as well as publish-subscribe mechanisms. However, it must be taken into account that the epidemic routing protocol does not guarantee the reliability in message delivery.

AGAPE [4] is presented as Another Message Oriented Middleware specially designed for MANETs. This collaborative middleware provides group membership and message-oriented communication in pervasive environments. It also offers context information of co-located group members, such as their attributes and characteristics. AGAPE organizes members in locality based clusters and considers two different roles depending on the device features: the cluster head and the managed entities. Low-resource devices like mobile phones or PDAs act as managed entities and rely on a more powerful device like a laptop, which would act as cluster head. Whereas this static role differentiation could be useful for team operations like emergency scenarios, we believe that is not specially suited for scenarios where collaboration between members is highly decentralized.

The next two reviewed solutions belong to the category of Peer to Peer Based Middleware. This kind of middleware utilizes a P2P communication model that involves resource and information sharing in order to perform a common task. P2P architectures share many similarities with MANET environments [5]: decentralization, dynamicity, and self-adjusting behavior. Hence, an association of both systems is believed to benefit the global operation of a collaborative application. However, most P2P systems were designed for wired and fixed infrastructures so adaptations are needed to use a P2P architecture in a infrastructure-less environment. Among the attempts to adapt P2P systems to mobile ad hoc networks, we now describe two middleware approaches that claim to offer an application framework: JMobiPeer and Peer2ME. The first one, JMobiPeer [6], is a JXTA compatible framework designed for J2ME CLDC environments. JXTA is the most mature P2P framework and provides interoperability and platform independence, allowing connection between heterogeneous devices. Hence, JMobiPeer benefits from these characteristics and introduces new features like a routing layer, emulation of multicast functionality to adapt JXTA to mobile environments, and the concept of code mobility. Nevertheless, JXTA based applications may introduce high communication overhead because the architecture does not take into account locality of nodes and relies on the exchange of XML messages.

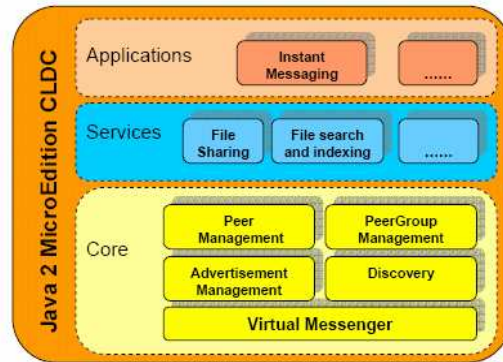


Figure 1 JMobiPeer Architecture

Finally, Peer2Me [7] is an application framework for mobile peer-to-peer applications which facilitates the development of this kind of applications and offers node discovery and messaging services. Several collaborative applications are provided together with the framework. It must be noted that Peer2Me is designed to be deployed on mobile phones under minimal J2ME configuration using Bluetooth devices. These last two frameworks only consider hand-held devices and therefore are not suitable for laptops or notebooks, where more complex applications could be deployed.

After evaluating these approaches, we can conclude that, to our knowledge, currently there is not an integrated solution which provides such a rich set of functionalities, together with the possibility of developing applications for the MANET environment in a fast and simple manner. Furthermore, it seems not easy to adapt any of the previous approaches in order to offer all the needed services to the upper layer. Because of this, next section will focus on evaluating a toolkit for building collaborative applications.

2.2.1.1. COMMUNICATION TOOLKIT: JGROUPS

JGroups is a reliable group communication toolkit based on IP multicast but extended with reliability and group membership. JGroups provides a flexible protocol stack architecture in order to suit each application with its specific needs. It can be used to create groups of processes whose members can send messages to each other. Group members can be spread across LANs or WANs. JGroups supports group creation and deletion, joining or leaving of groups, membership detection and notification, detection and removal of crashed members. Group members can send member-to-group messages or member-to-member messages.

The key characteristic of JGroups is that the reliability of multicast communication is a deployment issue, and does not have to be implemented by the developer of the application. The protocol stack can be extended with protocols that handle transport, fragmentation, failure detection, lost messages retransmission, ordering, membership and encryption. It is also possible to write and add a new protocol. A typical protocol stack could be the following:

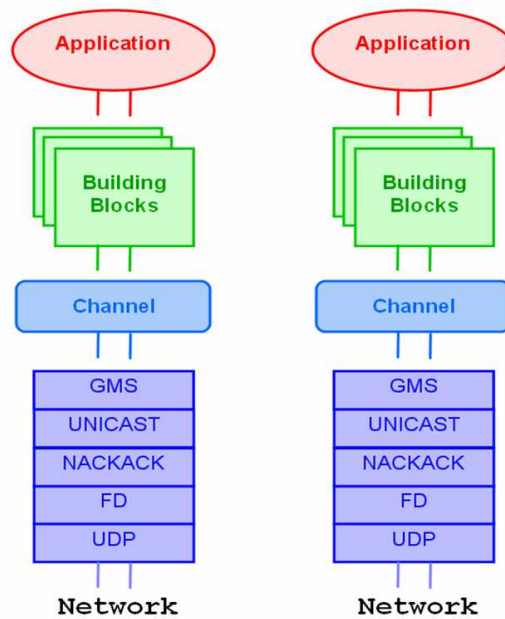


Figure 2 JGroups Protocol Stacks

All messages sent and received have to pass through the protocol stack. The protocols used in the example above are the following:

- UDP: IP multicast and unicast transport based on UDP.
- PING: Retrieves the initial membership multicasting ping requests.
- FD: Failure detection based on heartbeat messages.
- VERIFY_SUSPECT: Verifies that a suspected member is really dead.
- pbcast.STABLE: distributed message garbage collection.
- pbcast.NAKACK: Adds loss-less transmission and weeds out duplicates in multicast.
- UNICAST: Loss-less and FIFO delivery in unicast.
- FRAG: Fragments large messages into smaller packets. Reassembles fragmented packets into bigger ones.
- pbcast.GMS: Takes care of joining new members, handling leave requests and SUSPECT messages.

JGroups also provides high-level abstractions, called building blocks, such as a Replicated Hash Table (to create several instances of the same Hashtable in all group members), a Message Dispatcher (to provide synchronous communication) or an RPC Dispatcher (to invoke remote methods in all group members).

Although JGroups is not designed for high churn rate networks, it could work effectively in medium-size MANETs to provide a reliable communication channel.

JGroups-ME and JGroups CDC are two ports of JGroups for the J2ME architecture, the former for the CLDC (Connected Limited Device Configuration) / MIDP2 (Mobile Information Device Profile) and the latter for the CDC 1.0 (Connected Device configuration) / PP 1.0 (Personal Profile).

2.2.2. MULTIHOP NETWORKING

In this section we will review most important application layer multicast and analyze its main features. Since OMCAST is inspired by PAST-DM, we will explain this protocol in greater detail. However, in first place we will review shortly the two main unicast routing protocols for MANETs.

2.2.2.1. MULTIHOP UNICAST ROUTING

DYMO (Dynamic MANET On-Demand) is a reactive unicast protocol, successor of the popular AODV (Ad-Hoc On-Demand Distance Vector) and shares many of its functionalities. Routes are created on-demand by sending request and response control packets. In consequence, no global topology information is available. On the other hand, this means that when nodes stop sending messages, there is no overhead traffic in the network. DYMO seems more suitable for sparse communications, with mobility but may not work as well in case of congestion.

OLSR is a proactive unicast protocol, so it maintains routing table information up to date continuously. Topology information is exchanged by means of controlled flooding of topology messages. Hello messages provide information about the two-hop neighborhood in a way that each node selects a neighbor as MPR (Multi-Point Relay). These MPRs are in charge of sending topology messages to the entire network performing controlled flooding. With the topology information, each node can build the routing table in order to be able to send messages to the other nodes. OLSR performs well in small-medium sized networks where node density is relatively high. The knowledge of the topology, together with its good performance with dense communication patterns turns OLSR in a good candidate for performing group communication in MANETs. As OLSR floods the link state database unreliably, it may cause transient loops if the link state database becomes inconsistent due to packet loss.

2.2.2.2. APPLICATION LEVEL MULTICAST

AMRoute [8] was the first overlay multicast protocol proposed for MANETs. This protocol creates a shared tree for data distribution using only group members as nodes. The shared tree

is built, from a virtual mesh, with unicast tunnels that allow connection between group members. One of the main disadvantages of AMRoute is the static behavior of the virtual mesh, since no changes are made in the structure once it has been built. AMRoute does not handle network dynamics and leaves all responsibility for the underlying unicast routing protocol.

ALMA [9] (Application Layer Multicast Algorithm) creates a tree of logical links between the group members. The aim of this protocol is to reduce the cost of each link in the tree by reconfiguring the tree under mobility and congestion situations. When a node joins the network it must select a node as a parent, so as to become part of the tree. If tree performance drops below a defined threshold, the node must reconfigure the tree by switching the parent or freeing children. This mechanism leads to a complex loop avoiding and detection system, since synchronous switching can occur. ALMA also considers the existence of a rendezvous host for obtaining the structure of the logical tree as well as neighbor information in the bootstrapping process.

AOMP [10] (Ad-hoc Overlay Multicast Protocol) is an application-layer multicast that relies on reactive routing protocols to construct a delivery tree in a dynamic and decentralized way. This protocol proposes two stages: a first one that connects new nodes to the overlay and a second one that performs the tree construction and maintenance. AOMP takes advantage of the unicast routing protocol, and thus avoids routing overhead and improves scalability. However, this protocol is limited to use reactive protocols like AODV or DSR and only considers a single source node for the multicast session.

NICE-MAN [11] presents several improvements from the existing internet NICE protocol by exploiting the broadcast capability of the medium to reduce network traffic. The basic improvement is the maintenance of a reduced overlay formed just by cluster leaders whereas the rest of members are located one-hop away from at least one cluster leader. Thus, a node may send a message to various nodes simultaneously by benefiting from the broadcast nature of the medium. However, there are several drawbacks like the continuous selection of cluster leaders. Furthermore, non-overlay nodes are loosely connected since they do not send any control messages. This may imply high message loss, as nodes need to recover from the loss of connectivity. In consequence, membership information is not available either.

PAST-DM [12]; **Error! No se encuentra el origen de la referencia.** (Progressively Adaptive Subtree in Dynamic Mesh) is an overlay multicast protocol based on the construction of a dynamic virtual mesh. The mesh is maintained dynamically through the exchange of link state packets, thus adapting to network topology changes. These packets provide link state table information, that is, a partial view of the network. All nodes need to start the multicast session simultaneously, and afterwards initiate the bootstrapping process by sending TTL-bounded broadcast messages. With the topology information extracted from the mesh, nodes compute a

source-based Steiner tree to deliver information to all members in the multicast group. Logical and physical hop distances are used as heuristics to compute the Steiner tree. The source node takes its logical (virtual) neighbors as children in the tree. The rest of the nodes are packed into subgroups, which form a subtree where the root of this tree is one of the logical neighbors. Thus, each child of the source tree is responsible for delivering the multicast message to all nodes in the subtree. This process is repeated through every node until the subtree becomes empty. The decision of packet delivery path is computed at each receiver, so path selection is performed always with the most up-to-date information. Although this is an efficient way of delivering data, some packets may be lost if nodes change location, once the source node has computed its corresponding subtree.

The last protocol we will review is OMCAST [13]. This protocol, inspired in PAST-DM, presents a more flexible bootstrap procedure which allows hosts to join the multicast session at any time. OMCAST is specifically targeted for collaborative scenarios. It benefits from the broadcast nature of the wireless medium in order to minimize network delay and congestion. It also provides decentralized membership information which may become really useful for upper layers. This protocol has been developed here at URV together with the collaboration of Gerard París.

2.3. CONCLUSION

After analyzing both middleware and application multicast proposals, we have considered using JGroups as the substrate which all the middleware will be built on. On the one hand, JGroups seems to offer enough flexibility to provide all the required services in a fast and straightforward manner. The use of the communication channel allows reliable unicast and multicast delivery. The protocol stack also offers the opportunity to add additional functionality such as group membership or merge event information.

On the other hand, OMCAST stands as the best solution to provide multicast at the application level. OMCAST presents the most remarkable functionalities presented by PAST-DM and at the same time benefits from the broadcast delivery, hence reducing network traffic.

3. Architecture and Design

In this section, we will depict the main structure of the whole system. We will also show the overall architecture in terms of a first division in layers and modules, which may make a clearer distinction between the different relationships between all the functionalities. After that, each module is explained and detailed both in terms of functionalities and design.

3.1. OVERALL DESIGN

AGORA architecture can be defined in three main components. From bottom to up, the network layer, the middleware layer and the plug-in framework layer.

In first place, placed in the network layer, the application level multicast protocol is in charge of routing all multicast messages through the network. Considering topology and benefiting from broadcast nature of the medium, this protocol allows efficient packet delivery for group communication

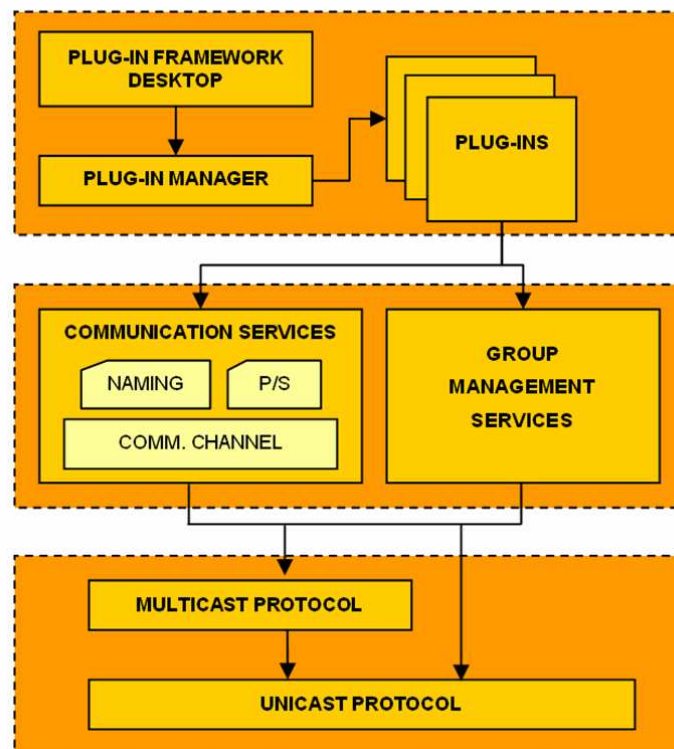


Figure 3 Agora architecture

Secondly, the collaboration middleware provides different communication mechanisms as well as group management primitives. Membership information, named communication channels and different communication paradigms are available for plug-ins. Finally, the plug-in framework allows fast development of applications as plug-ins. These plug-ins can be packaged, exchanged between peers and even be installed on runtime. This is achieved thanks to the plug-in manager, which also verifies integrity and controls plug-in lifecycle.

3.2. NETWORK LAYER

The network layer is the bottom-most layer and provides communication between all participants. Two basic primitives are available: unicast and multicast delivery. The network layer also provides mechanisms to solve problems as lack of ordering and unreliability of the delivery.

3.2.1. FUNCTIONAL DESCRIPTION

The network layer provides message delivery which will be used by communication channels. There are three well defined functionalities to perform communication through the network:

- Send a message to a single member of the group
- Send a message to all the members of the group
- Receive a message

Upon these basic functionalities, middleware services will build more complex communication paradigms, as naming and publish/subscribe services.

Besides, since this project is designed for collaboration scenarios, routing mechanisms must also consider its specific requirements and mobility model. As it has been stated in the requirements, group communication is the key for collaborative scenarios. By using an application level multicast (ALM), we ease the deployment of the whole application, so the user does not need to install a network layer multicast protocol. OMCAST, the chosen ALM protocol, specifically offers two main advantages:

- Broadcasting of data packets to 1-hop neighbors to reduce communication overhead.
- Decentralized membership information available for higher level layers.

OMCAST reduces global traffic thanks to the local broadcast delivery technique. The main idea of this process is to send just once as a broadcast message if there are enough neighbors located at one physical hop willing to receive the message. However, as almost all application multicast approaches, OMCAST does not guarantee reliable message delivery neither message ordering. Because of this, we need mechanisms to provide these two necessary functionalities.

Ordering and reliability

The network layer provides means to send unicast and multicast messages through the network. Since the use of TCP connections is discouraged, we have focused on ways of bringing ordering and reliability to the UDP protocol. These two features will be needed for collaboration at the middleware layer.

In order to provide reliability and ordering in a transparent and easy way, we took the toolkit JGroups as the foundation of our network layer implementation. In this project we have used the reliability and ordering protocols provided by JGroups. Since we are not using network layer multicast, we had to replace it by OMCAST, our application layer multicast, as it is depicted in Figure 4.

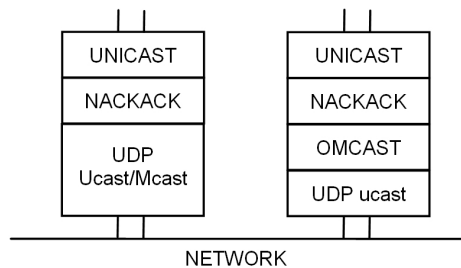


Figure 4 Protocol Stack Change

UNICAST protocol is in charge of FIFO ordering and reliability for unicast packets whereas NACKACK behaves in a similar way for multicast packets. UDP allows sending unicast and multicast messages in the original JGroups structure. As previously mentioned, in AGORA multicast packets are sent via OMCAST. In this way, we ensure that all application messages will be delivered to the receivers. Furthermore, configurations parameters such as number of retransmission, retransmission timeout and other can be easily configured and adapted to obtain a more appropriate behavior.

3.2.2. ARCHITECTURE

The structure is basically the one defined in the previous section. In first place, a unicast routing protocol is in charge of performing network communication. In our case, DYMO will be at the operative system level, so the packets sent and received by any application will be routed by DYMO. It must be remarked that by placing DYMO at the operative system level, we achieve transparency from the underlying unicast protocol. This could be useful if, for example we would like to operate in an ad-hoc network (no multi-hop). By removing DYMO we could simply use AGORA in the same way as it was used before, with no need of changing the framework.

Protocol Stack architecture

Once the messages gets to the AGORA framework the JGroups protocol stack processes the messages, so they pass through each protocol in the stack. In Figure 5 JGroups Structure and Protocol Stack we can see the structure of JGroups: The protocol stack offers different functionalities and allows the channel to send messages with specific properties: order, no duplicates, etc. Building blocks and applications built on top can benefit from the modularity

and flexibility of the protocol stack. A typical configuration of the stack would process the messages in the following manner:

1. UDP: the reception/sender protocol. Both unicast and multicast sockets are placed in here. In our case, we will only use the unicast socket, since multicast (ALM in this case) will use also the unicast socket.
2. UNICAST: unicast source-based ordering and reliability. All messages unicast contain a header with a sequence number that permit to ask for retransmission on lost messages. By placing this protocol just below OMCAST, it achieves that unicast messages sent by the protocol will also benefit from ordering and reliability.

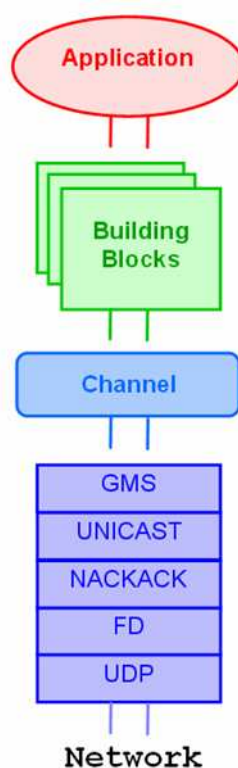


Figure 5 JGroups Structure and Protocol Stack

3. OMCAST: multicast sender/receiver. As a multicast sender, it receives multicast messages from the application and splits them in several unicast messages by constructing first a source-based tree. As a receiver, it collects one of this unicast messages and process them so the application recognizes messages as regular multicast messages.
4. MERGE: group merge. Merge is used after groups have split (they are far from each other and there is no connectivity between them) and they gain connectivity again. Coordinator election for GMS is performed and the merge procedure is notified through events to the applications.

5. NAKACK: multicast reliability based on GMS. Instead of using an ACK approach it uses the group view and asks for packets which have not been received (using negative acknowledgement).
6. GMS: provides a view of all the members currently connected to the group.
7. STATE_TRANSFER: when a new member enters a group, it provides a mechanism for asking for initial information to one of the members of the group.

3.3. MIDDLEWARE LAYER

The middleware layer includes communication and group management services. On the one hand, communication services offer the Naming Service, the Publish/Subscribe Service and the Communication Channel. These three paradigms provide the application with three different ways of communication. On the other hand, Group Management Services offer information about the group membership together with events which are fired when members join and leave the current group. Since we have decided to implement this platform under the Java technology, we have considered Java standards when implementing some of these services. Particularly, the Naming Service implementation is based on JNDI (Java Naming and Directory Interface) and the Publish/Subscribe service follows the JMS (Java Message Service) specification. Although the implementations only consider a subset of these interfaces, the use of standard APIs facilitates the development and maintenance of upper-layer services and applications.

3.3.1. FUNCTIONAL DESCRIPTION

Communication Channel

This module benefits directly from the primitives provided by the network layer. Furthermore messages have ordering reliability guaranteed due to protocols added to the JGroups protocol stack. These messages can be sent to a specific member or to all members of the group. The user of a communication channel can receive messages by invoking blocking calls or by managing message events that are fired whenever a message is received.

It is also possible to use a communication channel with a specific identifier that restricts the communication between channels with the same identifier used by different peers. These special channels are called Named Communication Channels. Messages that contain identifiers are filtered on reception and are delivered to the applications or services that have registered a listener with the specified identifier.

Provided functions:

- Create a default communication channel: a channel is created with a predefined name.
- Create a named communication channel: a channel is created with a specific name.
- Send a message: a message is delivered to a single member.
- Send a message to the group: a message is delivered to all the members of the group.
- Asynchronous receive: an event is fired when a message is received.
- Synchronous receive: blocking call. The application blocks until a message is received.

Naming Services

Each created group maintains a decentralized Naming Service which binds names with resources and locations. These resources can be any kind of information sources, registered plug-ins, or other available objects and services. The Naming Service is based on a Replicated Hashtable, i.e. a replicated structure provided by JGroups which replicate each entry of a hash table to all members of the group. The naming service contents are replicated among all members of the group; therefore these contents are automatically available to all members of the same group whenever they are generated. For this reason, the naming service is intended to store just lightweight information in order to avoid generation of excessive traffic.

The Naming Service provides an implementation of a subset of the JNDI specification which offers the following functionalities over a flat namespace:

- Bind a name to an object.
- Bind a name to an object, overwriting any existing binding.
- Unbind a specific object.
- Retrieve the named object.
- Enumerate the names bound in the named context, along with the objects bound to them.

When a new name is added, removed or the bound object contents are changed, the Naming Service fires a NamingEvent. This event can be handled by upper layer services or applications using dedicated JNDI listeners.

Moreover, when a new user joins the group, the current state of the Naming Service is automatically retrieved. The new user obtains the same view of the naming information that is shared by all the peers of the same group. In addition, when two groups merge back, a new view of the naming service is generated and is sent to all peers. This process does not need the intervention of the user because is automatically done whenever a peer joins an existing group or two groups merge back.

Publish/Subscribe Services

In every group, different publish/subscribe channels are available. In this way, users can create topics and then assign subscriptions to them. Subscriptions are needed to receive messages published under a certain topic. To be able to retrieve messages after network disconnection, durable subscriptions are also available.

The publish/subscribe mechanism uses the multicast capability of the JChannel to send a message to all members of a group. If a peer is not interested in receiving messages from a

certain topic, messages will be automatically discarded. Messages are not sent via the communication channel since this channel is intended to be used just by higher level services.

Durable subscriptions persist to network disconnection, so that if a peer leaves the network due to disconnection or mobility issues, messages published on that topic are delivered automatically to the subscriber when rejoining the group. This can be achieved since all members (with a durable subscription on a topic) keep all received messages for a certain time. Then, lost messages or previously posted messages can be recovered by asking and retrieving them from one of the members of the group.

This publish/subscribe functions are implemented as a subset of the JMS specification. Only the multicast model is implemented, which means that the point-to-point model is not available.

Provided functions:

- Create a topic: allows to send/receive messages on a topic
- Publish a message under a topic: sends a multicast message to all group members.
- Create a subscription to a topic: allows a member to receive messages published under a certain topic.
- Create a durable subscription to a topic: allows a member to receive messages published under a certain topic and to keep them for a certain period of time.
- Register a listener to process messages: defines how messages will be processed when they are received under a certain topic.

Group Management Services

Group management services support multiple group creation. Groups are essential for collaboration, since all actions are performed inside the scope of a group.

Groups are created by providing the group name and then, multicast address and port mapping are provided from the underlying services. Each member can join several groups at the same time and they will be notified about event generation whenever other groups are created or deleted. Membership on each group is provided as a list of members that currently belong to the group, together with notification about member join/departure events. Furthermore, reconciliation after network merge will be available for services such as naming and publish/subscribe.

Provided functions:

- Get group members: returns a list with all the members in the current group.
- Add listener for membership changes: allows registering a listener into a group that notifies changes whenever a members joins or leaves the group. These listener methods will be invoked when a Join/Leave event is produced.
- Remove a listener for membership changes: remove one of the previously registered listeners.
- Add listener for group membership changes: allows registering a listener that notifies changes whenever a group is created or deleted. These listener methods will be invoked when a *groupCreated* or *groupDeleted* event is produced.

3.3.2. ARCHITECTURE

In the following section, we depict the different modules comprised in the middleware layer. Since this is the more complex layer, class and sequence diagrams are provided together with a brief explanation of each process performed by the middleware modules.

Communication Channel

Communication Channel static view

The Communication Channel provides basic communication primitives to send and receive messages and can be used by applications or upper-layer services. It benefits from the underlying JChannel to send both unicast and multicast messages in a reliable and ordered manner. CommunicationChannel do not use JGroups functions directly but an adapter component (CommunicationAdapter). This component adapts all communication functions to the JGroups API and may be modified if the underlying components are changed.

Communication Services dynamic view

The basic behavior of the Communication Channel is depicted in the following sequence diagram (Figure 6). The user (applications or upper-layer services) wants to create a communication channel and sends a request to the Group (1). The application can specify a name for the channel, thus allowing multiplexed channels over the same group. The Group class is in charge of creating new communication channels (2) and returns the new channel to the application (3). Then, the application can register its own listeners to the channel (4), in order to receive messages sent to the channel. Received messages addressed to a channel that has been registered in the CommunicationChannel are automatically delivered to the registered listener.

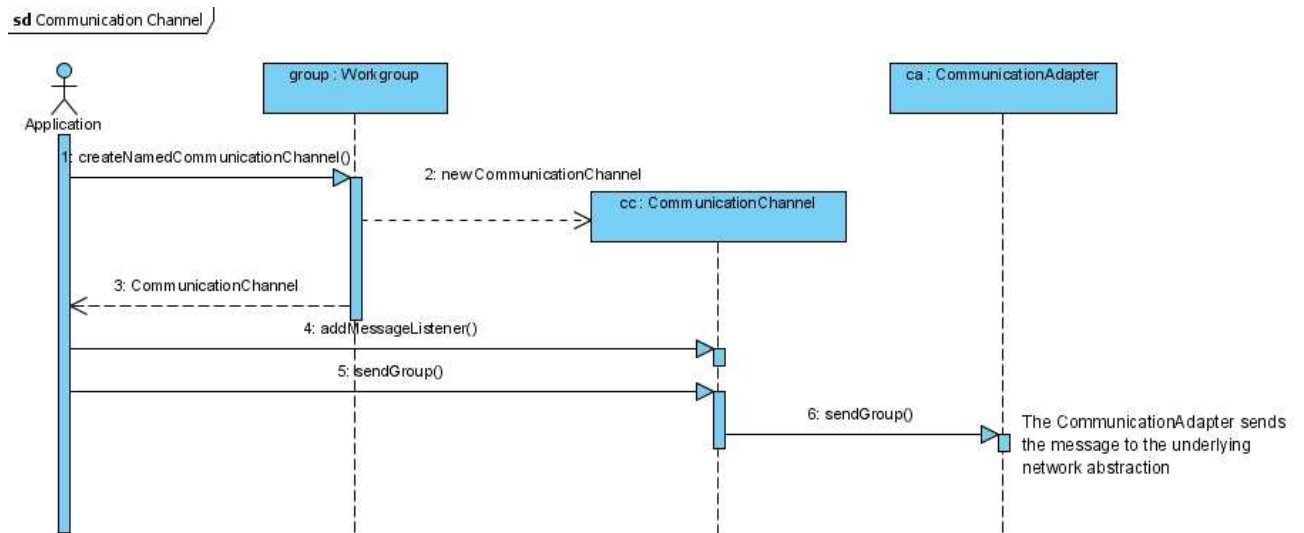


Figure 6 Communication Channel Sequence Diagram

The application can also use the communication channel to send messages to all the members of the group (5) or to a specific member. The CommunicationChannel accesses the CommunicationAdapter (6) to forward the message to the Network Abstraction Layer.

CommunicationChannel also provides a function to enable synchronous receiving. When this receiving method is enabled, messages addressed to the channel are delivered to registered listeners and can also be obtained by invoking a blocking receive function. The time to wait for a message is configurable and if no message is received in this time, the CommunicationChannel raises a TimeoutException.

Naming Services

Naming Services static view

The naming service architecture is based on the Java Naming and Directory (JNDI) specification, which is a standard Java interface to build naming and directory services. By using a well-known standard, we provide an easy-to-use module to the upper layers of the architecture. The main component of a JNDI implementation is the Context, an interface which provides access to a set of resources bound to names. All the registered resources are stored in a data structure provided by JGroups called ReplicatedHashtable. Context maps all modifications of the naming space to the ReplicatedHashtable, which is also responsible for communicating the modifications to all the members of the group. The ReplicatedHashtable is, basically, a transparent replicated structure among all the members of the group.

Moreover, JNDI provides standard events (NamingEvent) to notify applications of the changes that take place in the Context: name addition, name deletion and changes in the

bound resources. Applications have to implement a NamingListener in order to receive such events.

Each Naming Service, as well as other middleware services, is attached to a specific group. Therefore, the group component must provide a function to obtain its associated Context. This function returns a factory (FlatInitCtxFactory) that allows the creation of an initial context for the current group.

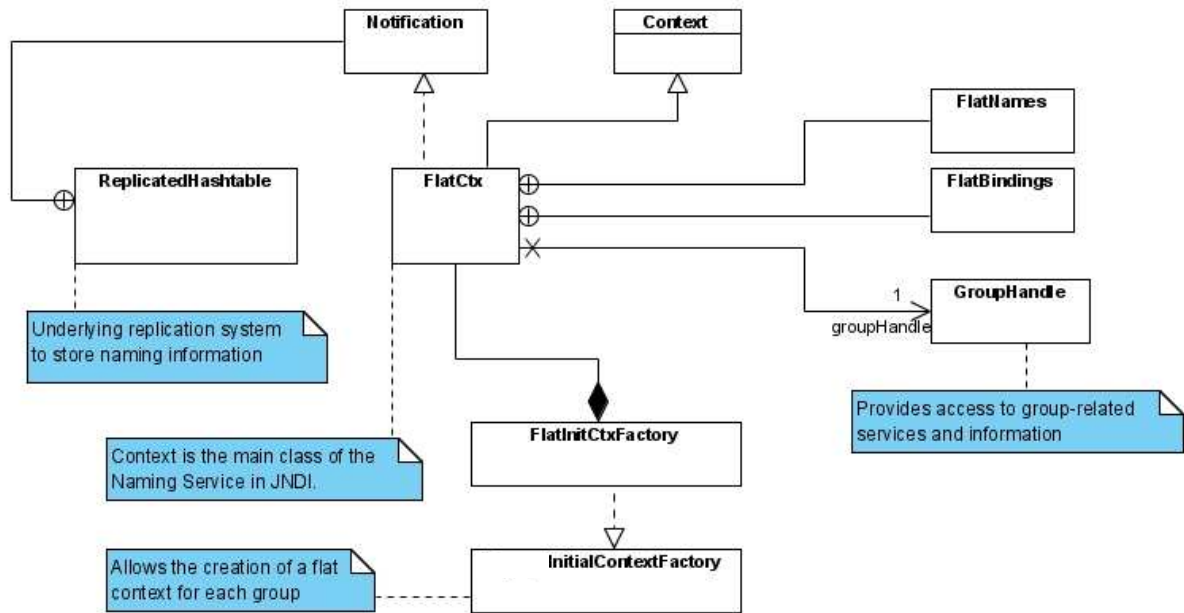


Figure 7 Naming Services Class Diagram

As it is depicted in the Figure 7, the Naming Service provides event notification when a registered entry changes or a new entry is added. The Context implementation consists in a flat context without name hierarchies (FlatCtx). The naming information is replicated among the peers of the group, i.e. all the peers have the same updated information. The underlying mechanism to ensure this replication is the ReplicatedHashtable, which is in charge of notifying naming changes invoking the methods provided by the Notification interface.

The InitialContextFactory class provides methods to create the Naming Service for each group of peers. This Naming Service can access to group-related services and information through the GroupHandle component, e.g. to register listeners that allow receiving messages addressed to the NamingService.

Naming services dynamic view

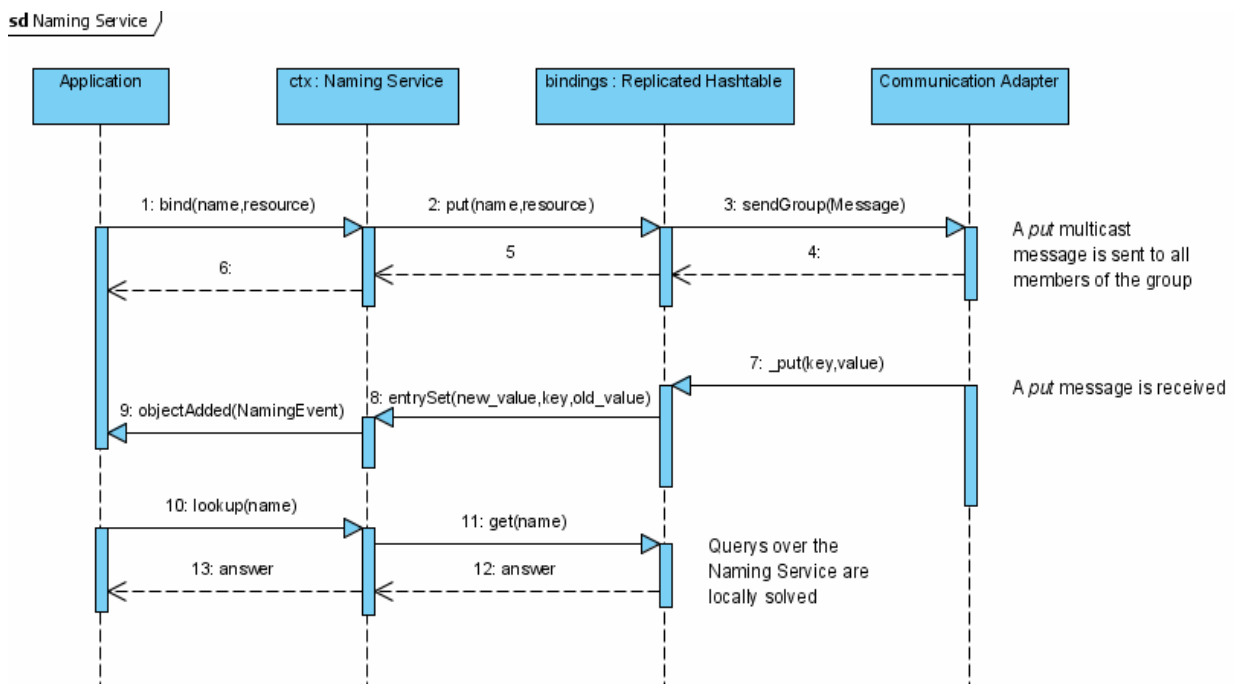


Figure 8 Naming Services Sequence Diagram

Figure 8 shows the Naming Services sequence diagram. When an application or an upper layer service that use the naming service wants to register a resource, it invokes the bind method on the naming service (1). As the naming service is based on a replicated hashtable, a put operation (2) must be invoked on the replicated data structure, which, in turn sends a *put* message to all the group members (3).

When a *put* message is received, the replicated hash table receives the corresponding event (7) and updates its data structure with the new information. It is worth remarking that any peer could be the source of the received multicast message, including the same peer that receives the message. Once the Naming Service receives the corresponding notification event, it is in charge of creating and dispatching a new detailed event (8). This NamingEvent may include the new name and its old and new bound object, if they are available. Applications that have shown its interest in a certain view of Naming Service changes will receive this event.

Related operations like rebind and unbind present a similar behavior, spreading the modifications to all members of the group using multicast messages. On the other hand, the diagram shows the behavior of the lookup operation. This operation returns the resource bound to a certain name. When the Naming Service receives this type of query, it performs a get operation on the ReplicatedHashtable, which returns the answer if available. So, all queries over the Naming Service are locally solved and do not initiate a network operation.

Publish/Subscribe Services

Publish/Subscribe Services static view

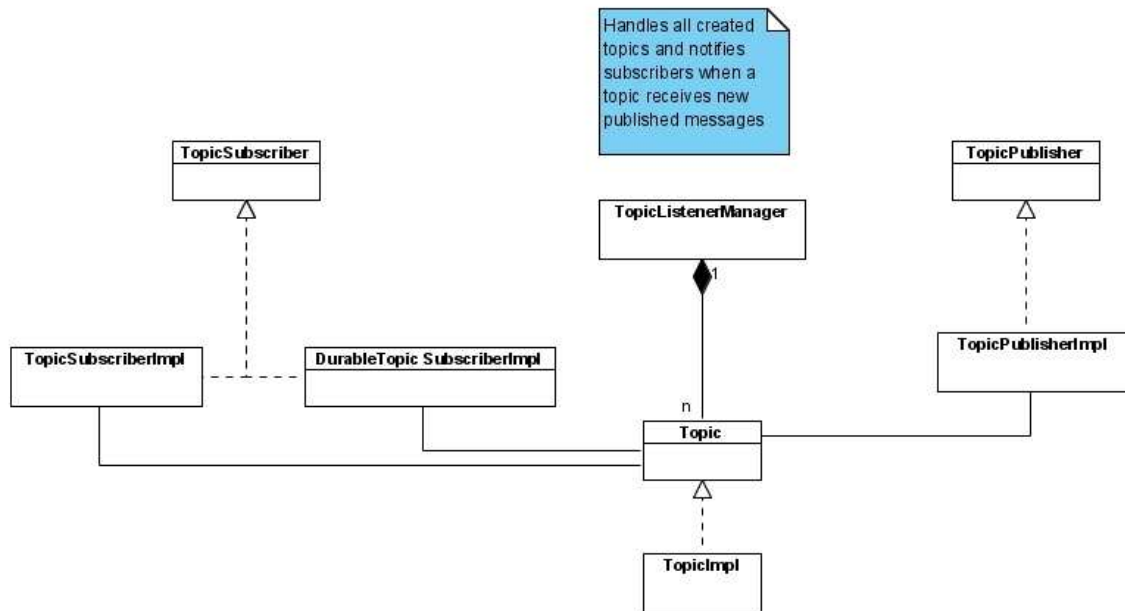


Figure 9 Publish/Subscribe Services Class Diagram

The Publish/Subscribe service benefits from the JMS interface to provide a topic-based publish/subscribe mechanism. Since JMS is not fully integrated with JNDI, the TopicConnectionFactory is obtained from a group. Then, the TopicConnection and the TopicSession can be obtained in order to create subscribers, publishers and topics.

On the one hand, all messages are sent through a Topic via a TopicPublisher. On the other hand, in order to receive messages, two different kinds of subscribers can be bound to a Topic. TopicSubscribers just receive the messages whereas DurableTopicSubscriber also keep received messages. Thus, messages can be transferred to other peers in case they need to ask for previous messages when they rejoin the group. The topic abstraction is represented by the Topic class. This TopicListenerManager is in charge of keeping all registered listeners from durable and non-durable subscriptions. Therefore, this manager maintains a list of all existing topics and notifies the subscribers when a new message is published on a specific topic.

Publish/Subscribe Services dynamic view

There are three basic operations performed in this module. The first two operations are referred to basic communication functionalities in the publish/subscribe service.

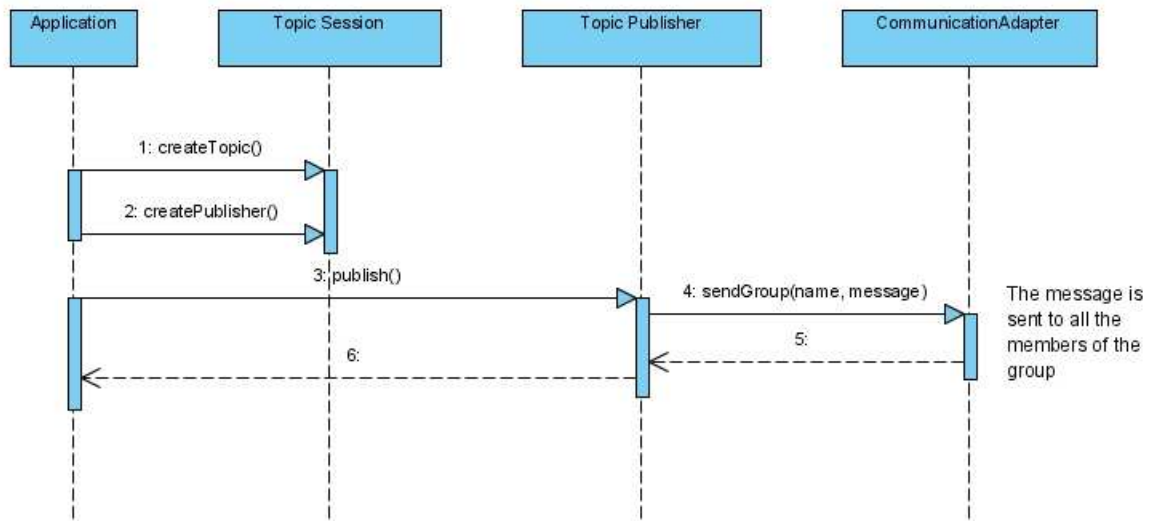


Figure 10 Publish/Subscribe Services sequence diagram: publish

When a publisher wants to be created, the application requests the TopicSession for a new publisher (1 and 2). When the topic is bound to the publisher, messages can be sent by invoking publish (3) on the selected publisher. The message is forwarded to the communication adapter, which sends a multicast message (4), with the topic identifier, to all members of the group.

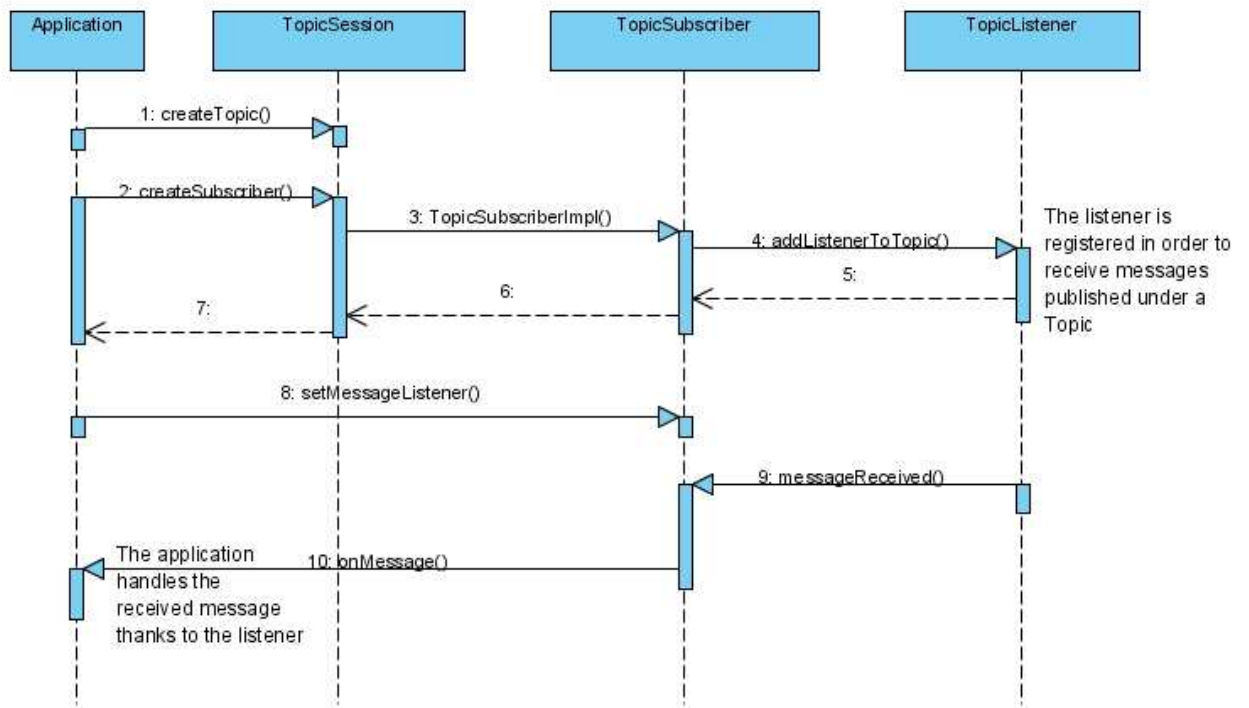


Figure 11 Publish/Subscribe Services sequence diagram: subscription

On the other hand, messages sent by TopicPublishers are received just for TopicSubscribers. Similarly to the previous case, the application requests the TopicSession for Topic (1) and TopicSubscriber (2) creation. Then, the subscriber registers a listener in the given Topic (3,4), so it is able to receive messages. However, messages are not delivered yet to application until a message listener is bound to the subscriber (8). Therefore, when a message is received (9), the TopicListener forwards the message to the listener registered in the TopicSubscriber, so the MessageListener is invoked (10).

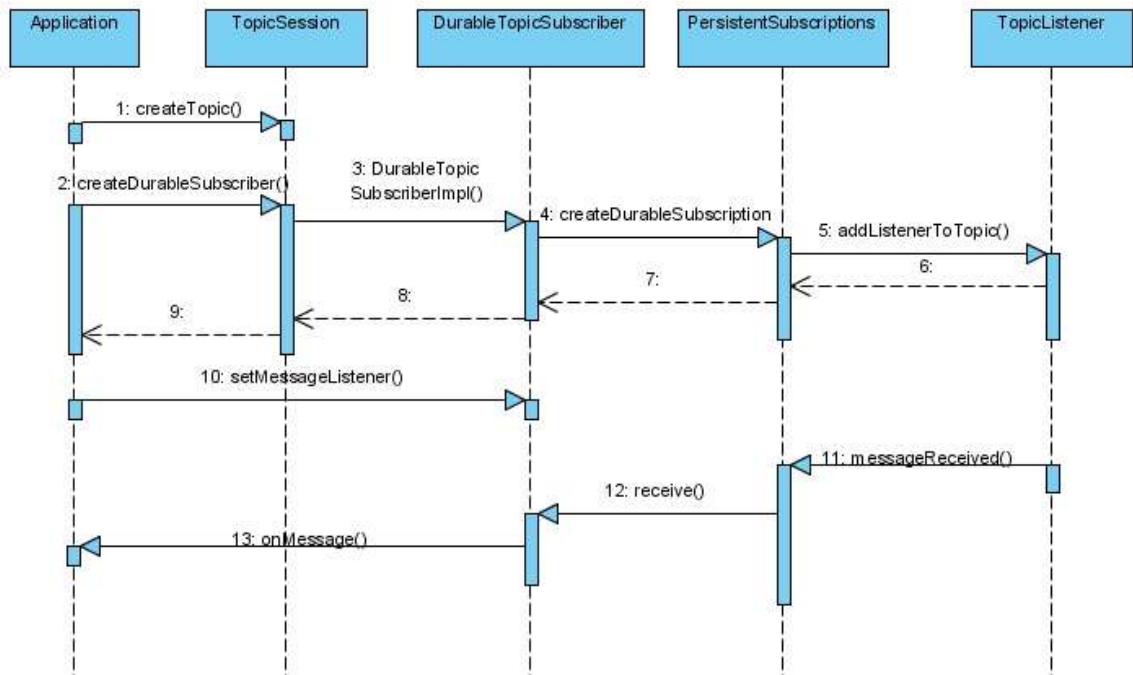


Figure 12 Publish/Subscribe Services sequence diagram: durable subscription

Moreover, DurableTopicSubscriber can be used to avoid losing messages on network disconnection. The behavior is similar to the TopicSubscriber, except for the PersistentSubscriptions class, that keeps all received messages in case they are necessary for other members when they rejoin the group.

Group Management Services

Group management has been built taking the group class as a foundation of the architecture. Several groups can be created simultaneously, which will be managed by the GroupManager. Then each group offers access to the underlying services: communication channel, naming service, etc. Furthermore, the group handle encapsulates all structures needed for group operation.

Group Management Services static view

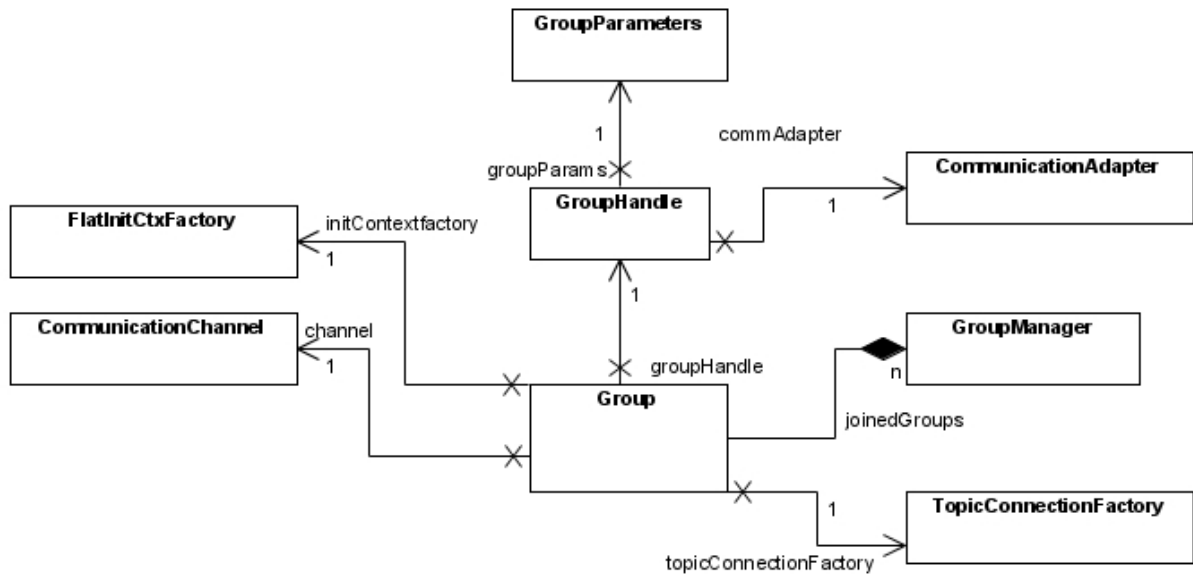


Figure 13 Group Management Services class diagram

As it is depicted in the above class diagram (Figure 13), the Group Management Service is handled by the GroupManager class. This class maintains an updated list of all joined groups by the local peer. The Group class is an abstraction of a group of peers which includes basic group functionalities such as FlatInitCtxFactory (used to create the Naming service), the CommunicationChannel (basic communication primitives) and the TopicConnectionFactory (publish/subscribe services). It is worth noting that all the middleware services are always tied to a specific group of peers.

The GroupHandle class is an auxiliary class that maintains specific information of the group (GroupParameters) and gives access to the CommunicationAdapter, the class that provides interaction with the network abstraction. Messages received through the publish/subscribe system are handled by the TopicListenerManager.

Group Management Services dynamic view

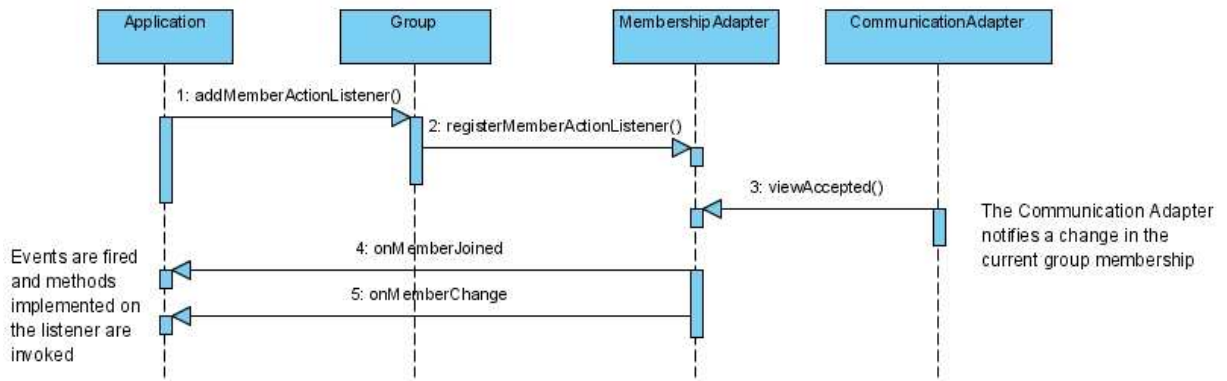


Figure 14 Group Management Services sequence diagram: event notification

Group management offers event generation whenever a member joins or leaves the group. In first place, when the underlying membership service offers a new view of the group (3), the MembershipAdapter computes the events to be triggered (4, 5). Then, the event is forwarded to the application, which previously registered a MemberActionListener (1, 2).

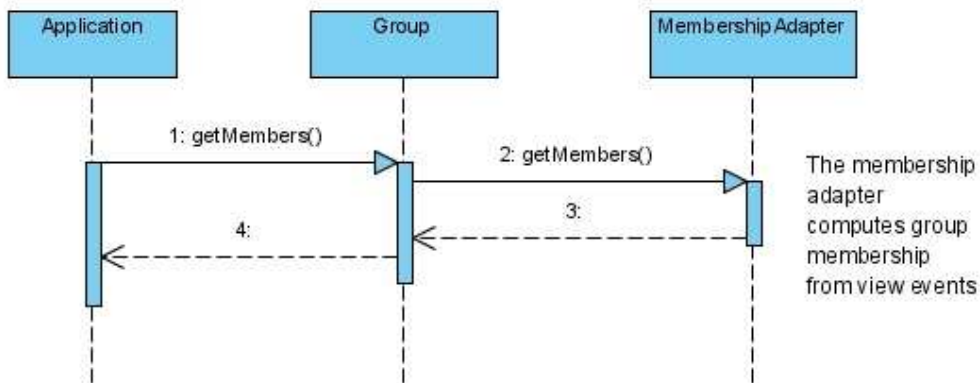


Figure 15 Group Management Services sequence diagram: obtain members

When information about group members wants to be retrieved, the application (or another service) asks the group (1) which forwards the query to the membership adapter (2). Finally, the information is returned to the application.

3.4. PLUG-IN FRAMEWORK

The plug-in framework is used as a proof of concept of the two underlying layers, both the network and middleware layer. This layer is composed of three different modules which allow the user to interact and use the middleware functionalities:

- The plug-in module represents the applications the user can interact with (so in fact, many modules instances are present). Plug-ins use the middleware functions to provide message-exchange, file sharing and many other services to the end-user.
- The plug-in manager is in charge of controlling the plug-in lifecycle. The manager starts and stops the plug-ins when requested. The manager may download plug-ins from other peers and also verifies the format of the plug-ins to be correct.
- The plug-in framework desktop corresponds to the graphical interface that allows the user to start, download and use the plug-ins.

3.4.1. FUNCTIONAL DESCRIPTION

Plug-in module

Plug-ins are the final applications that benefit from the underlying services that the middleware layer provides. Plug-ins must follow a specific interface in order to be loaded into the system. Each implemented plug-in must conform to the Plug-in Interface, which allows the framework to:

- Obtain the Container of the plug-in, so it can be placed into the framework desktop
- Start the plug-in and stop the plug-in. Initialization and finalization methods must be set here.
- Register a plug-in listener, so the plug-in manager can invoke lifecycle methods (start/stop). The inversion of control allows the manager to launch the plug-ins when requested by the user.
- Set the Session instance. As it will be explained hereafter, it provides access to middleware services.

Session is a well-known API that defines which services are available for plug-ins. Services like the Communication Channel, Publish/Subscribe and Naming Services are available so the plug-ins have access to a rich but restricted subset of functionalities. The methods in the API provide the following functionalities.

- Create and destroy communication channels

- Obtain the TopicConnectionFactory which gives access to the Publish/Subscribe services (JMS)
- Obtain the Naming Context which gives access to the Naming Services (JNDI)
- Obtain information about the current membership, i.e. get all the members which are currently connected to the group. It is also possible to register listeners that notify changes in the membership.

Then, in order to create a Plug-in it is only necessary to follow the Plug-in interface, and the all needed functionalities will be available from the Session object.

Plug-in manager

The plug-in manager is in charge of all the tasks related to plug-in management:

- Launch the plug-ins that are currently installed in the framework.
- Perform plug-in file verification before launching the plug-ins, correct file descriptor, plug-in name, etc.
- Collect information about plug-ins that are shared in other peers. This information is made visible through the plug-in framework desktop, so the user can retrieve plug-ins that are not installed yet from its machine.

Plug-in framework desktop

The plug-in framework desktop basically gives access to the functionalities provided by the manager, in this case through the graphical user interface. Besides, it provides other functionalities:

- Launch plug-ins without having to restart the framework.
- Visualize several plug-ins at the same time.
- Show a list that allows launching a specific plug-in.

3.4.2. ARCHITECTURE

Plug-in module

Plug-in module static view

In Figure 16 we can see in the typical architecture of a plug-in, in this case, the Chat Plug-in. The plug-in implements the Plugin interface, and at the same time provides a Session to the Plugin, so it can access all the middleware functionalities.

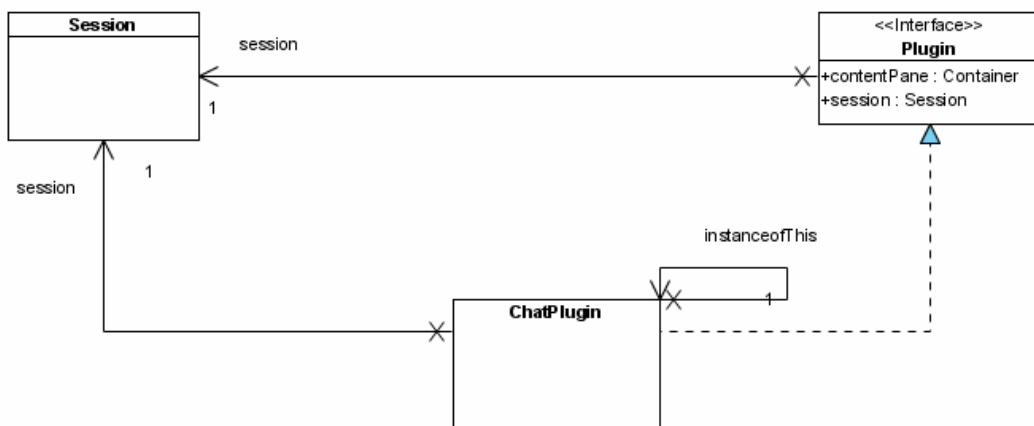


Figure 16 Plug-in module class diagram

Plug-in packaged files

Plug-ins, once they are implemented, they must be packaged in order to be installed into the system. In this section will depict briefly how a plug-in is packaged and what are the requirements that an AGORA plug-in needs to meet.

In first place, all classes referenced by the plug-in (i.e. all “new” classes specifically created for the plug-in to work) must be included into a file. This file, usually a compressed file, must be a jar file (must contain the jar extension in the filename) and also contain a plug-in descriptor as the following:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<Plugin>
  <Name>urv.test.plugin.swing.anagramplugin.AnagramPlugin</Name>
  <Author>URV</Author>
  <Description>This plug-in allows the user to play the anagram
game</Description>
  <Version>0.1</Version>
  <Icon>images/letters.png</Icon>
</Plugin>
  
```

The “name” tag indicates the main class of the plug-in (the one that implements the Plug-in interface) to be launched. The “description” and “version” tag will be shown in a small box when a user wants to list the existing plug-ins in the system. It is also possible to add an icon that will be attached to the plug-in button in the toolbar, so that the plug-in is easier to identify.

Plug-in manager

The PluginManager class uses the CustomClassLoader in order to load the necessary classes from the plug-in. The PluginListener is invoked by the user when he wants the plug-in to be stopped. This is important since the manager is the one controlling the plug-in lifecycle, and

is the only class which can start or stop the plug-in. The plug-in manager has also access to the plug-in framework desktop via a registered listener, so the manager may force user messages to appear, or modify the desktop appearance when the plug-in status has changed.

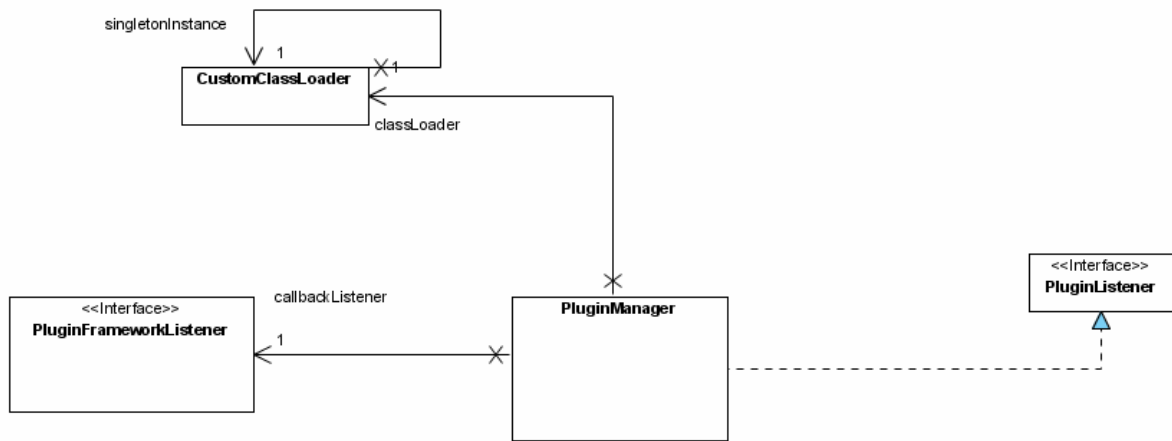


Figure 17 Plug-in manager class diagram

Plug-in framework desktop

The PluginFramework class uses the PluginManager to perform all needed operations on plug-ins. Apart from this, the PluginFrameworkDialogs class helps the framework in showing and having feedback from the user (input and output operations) and the PopeyeStyle class defines the look and feel of the FrameworkDesktop

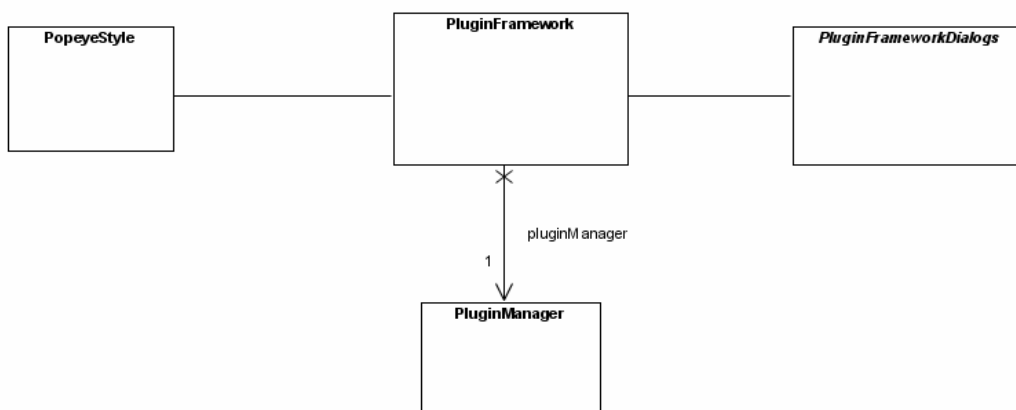


Figure 18 Plug-in Framework Desktop class diagram

4. Development

The development of AGORA has been an iterative process in which different parts of the framework have not been implemented concurrently. The development started on the middleware layer, since it is the most important layer of the architecture. To test the correct behavior of the middleware we used ad-hoc configurations of the network layer to perform communication on Ethernet networks. Once the tests were successful we implemented the plug-in framework desktop, and we left the implementation and the adaptation of the MANET multihop protocols as the last stage of development.

First stage of development

Initially, middleware services were the first to be implemented so they could be tested exhaustively through all development process. Basic group management was available: group creation, listing the members of a group without event notification, etc. Both JMS and JNDI implementations (publish/subscribe and naming) presented basic functionalities: topic creation with non-durable subscriber and naming storage information (no search available). JGroups protocol stack was configured to work in Ethernet and ad-hoc networks (no multi-hop).

Then, it followed the design and implementation of specific test for each middleware module and network communication: CommunicationTest, JNDITest, JMSTest, as well as multicast tests to verify that multicast communication is enabled on the participant nodes

Second stage of development

After the middleware was tested thoroughly, the basic plug-in framework implementation started. No remote plug-in download was available, although plug-ins were installed from the local file system or downloaded from an URL. As it can be seen in the figure a small port for PDAs, more specifically for Java CDC 1.1, was also implemented. Both laptops and PDAs could communicate with the two first plug-ins: Chat Plug-in and Shared Photo-Album. The implementation of the services from the middleware layer was almost finished. Design of the multicast protocol for multi-hop network protocol started.

Last stage of development

All functionalities were available, remote plug-in download implemented and the rest of plug-ins were also available. During this stage we developed the application level multicast protocol, OMCAST¹, as a JGroups protocol, so now multicast was also supported under a

¹ This protocol has been developed together with Gerard Paris Aixalà. For more information, please refer to the Annex section.

multihop network. The whole application was tested with DYMO (in Linux) and OLSR (in Windows).

5. Evaluation

We have developed different standalone tests as well as multiple plug-ins in order to validate the correct behavior of the middleware services and the plug-in framework. Apart from this, the middleware layer has been used in the IST Popeye project as a basic services layer which provides communication and group services to context-awareness, security and core services modules.

Plug-ins

Plug-ins are the applications that use middleware functionalities. Hence, the set of developed plug-ins are intended to demonstrate as well as to evaluate the different capabilities of the middleware. Here is the list of developed plug-ins:

- Chat plug-in:

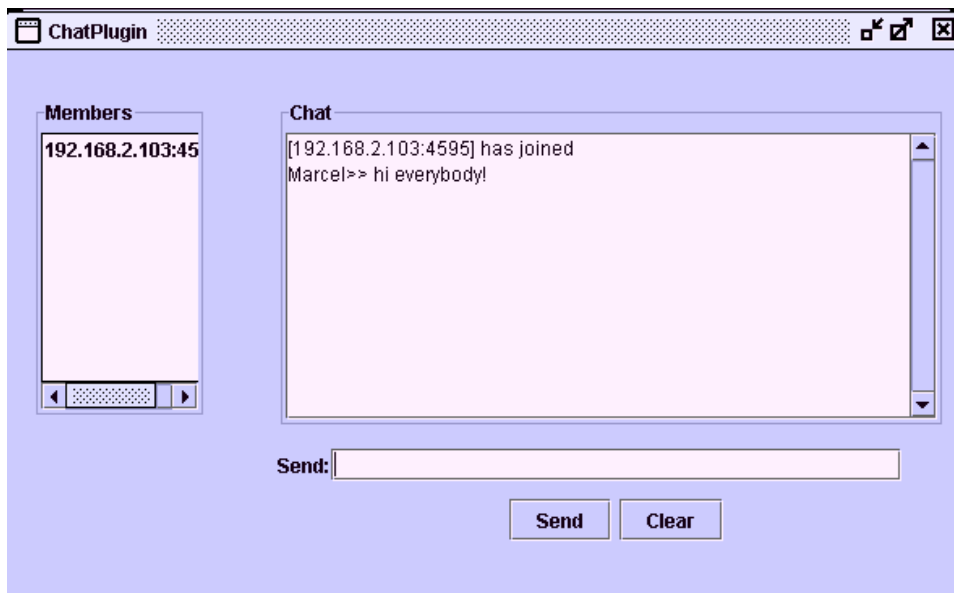


Figure 19 Chat Plug-in

The Chat plug-in allows users to communicate by exchanging group messages or direct messages. Users can send group messages in the main screen, similar to a chat room, so all users will receive the message. By double-clicking on the IP address of a chat participant, a new window will appear to enable private messaging between two participants.

Tested services: Communication channel, multicast and unicast delivery.

- Shared album plug-in:

Multicast plug-in permits sharing photos between different users. Users can browse their local drive to search for photographs. After choosing one, they just have to assign a name to it. Then, information about the new resource and the location of the

resource are stored in the naming service. The plug-in gets notified whenever new objects (resources) are stored in the naming service. By clicking on the “Show” button, users can download the shared photograph to their machine.

Tested services: Communication channel, unicast delivery, naming services with naming events.

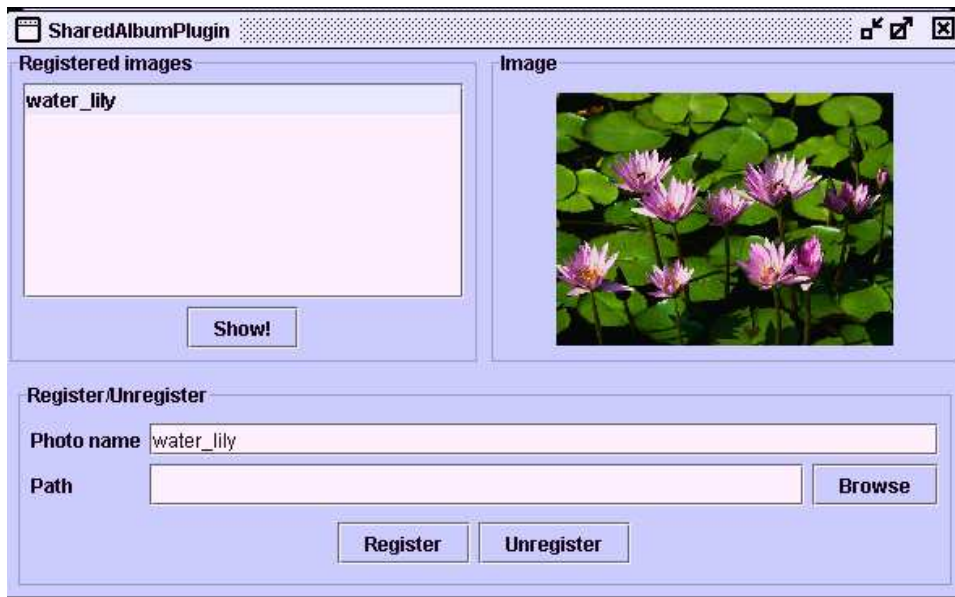


Figure 20 Shared Album Plug-in

- Anagram plug-in

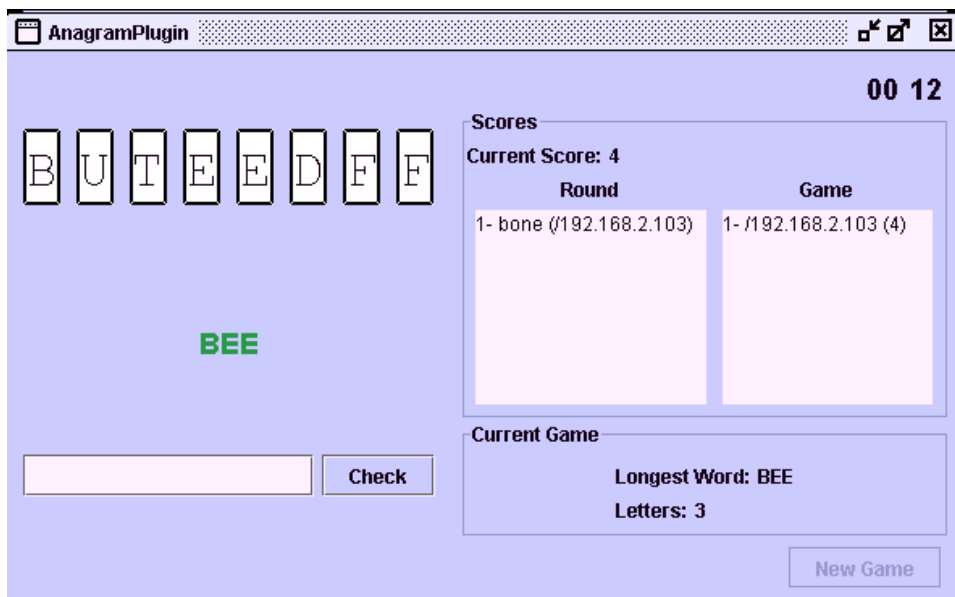
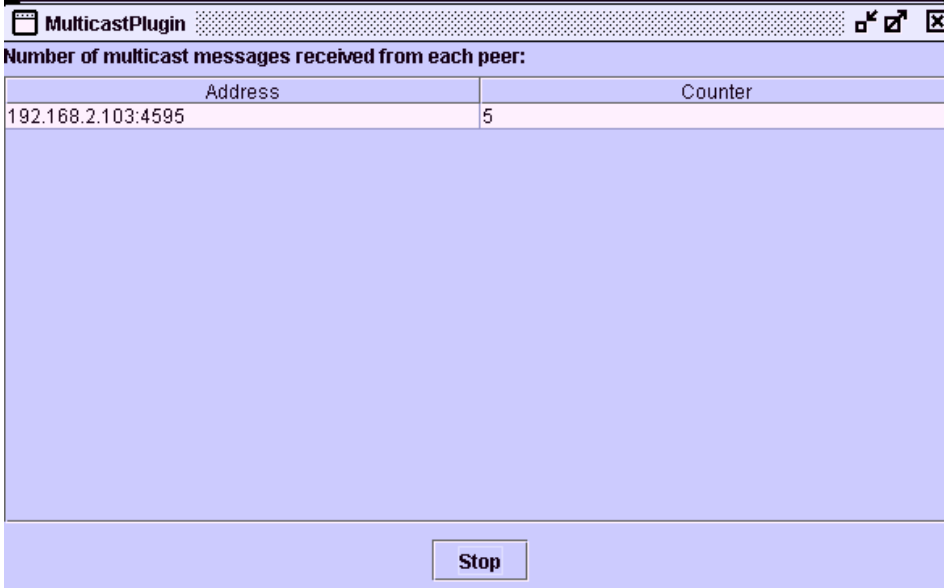


Figure 21 Anagram Plug-in

The anagram plug-in is a competitive game plug-in. Users must form words with the given letters. There is a maximum of 30 seconds to form the longest possible word. The longest the word is, the more points the user gets. The word is spread via a multicast message as well as the results of each round.

Tested services: Communication channel, multicast delivery.

- Multicast plug-in:



The screenshot shows a window titled "MulticastPlugin" with a table titled "Number of multicast messages received from each peer:". The table has two columns: "Address" and "Counter". One row is visible with the address "192.168.2.103:4595" and a counter of "5". A "Stop" button is located at the bottom center of the window.

Number of multicast messages received from each peer:	
Address	Counter
192.168.2.103:4595	5

Figure 22 Multicast Plug-in

Plug-in used for testing correct multicast message delivery. Most used when developing the application level multicast protocol. The plug-in counts how many multicast messages have been received from each multicast source.

Tested services: Communication channel, multicast delivery.

- News plug-in:

This plug-in utilizes the publish/subscribe capabilities of the TopicConnectionFactory. Messages are published under a default topic, so that users can leave and rejoin the group and they will receive all messages published during the disconnection.

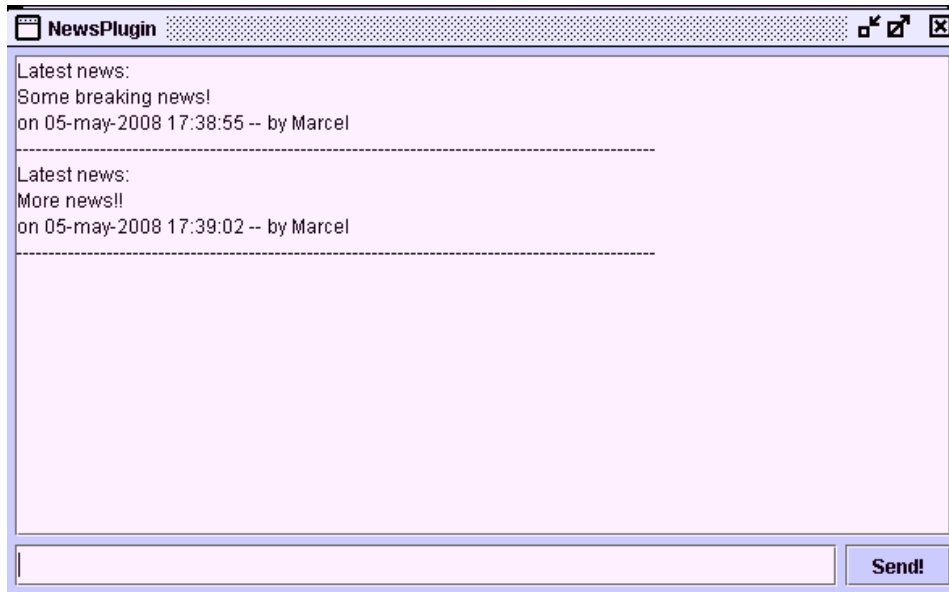


Figure 23 News plug-in

Tested services: Publish subscribe services, multicast delivery.

- RTT plug-in:

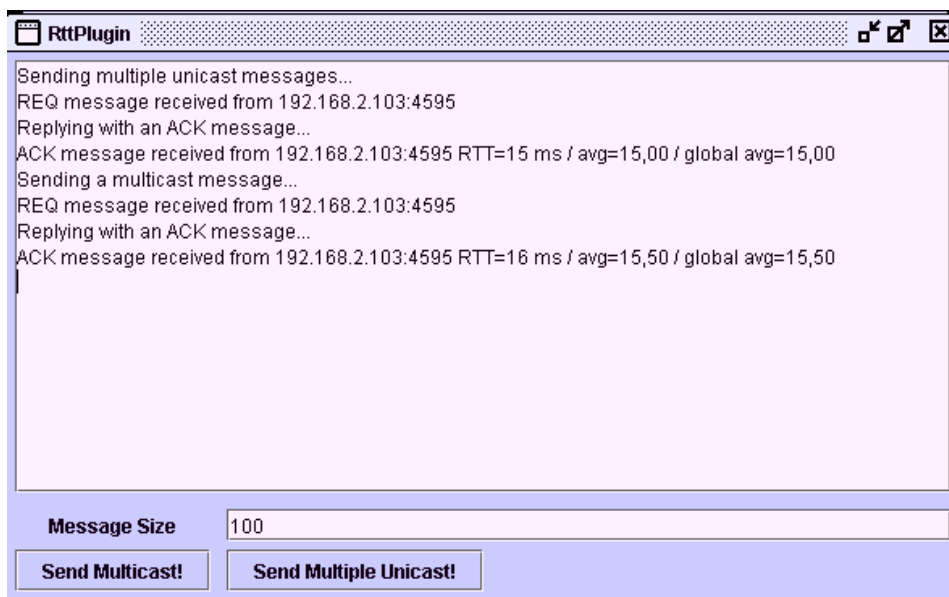


Figure 24 RTT Plug-in

This plug-in computes the mean round trip time (RTT) of sending multicast messages and receiving a response via unicast. It is also possible to establish the size of the message and change multicast delivery for multiple unicast in order to test multicast performance.

Tested services: Communication channel, multicast delivery, unicast delivery.

Standalone Tests

These tests perform individual checks of very specific functionalities of AGORA. They have been usually used in the early stages of development.

1. **CommOrderSenderTest**: sends a certain number of messages to a specific group. Messages are sent with a fixed delay, and they should be received by a different application which joins the same group and creates a communication channel with the same name.
2. **CommOrderReceiverTest**: the receiver application for the previous sender application.
3. **CommunicationTest**: creates a named communication channel and sends group and unicast messages to all members of the group. It also checks membership in the local group and in all the groups.
4. **ContextEventsTest**: adds `NamespaceChangeListener`, `ObjectChangeListener` to the context (naming service) and checks behavior when events responding to these actions are fired.
5. **GroupTest**: the local node joins several groups and group membership as well as global membership is checked.
6. **JMSTest**: gets the `TopicConnectionFactory` to test publish/subscribe functionalities. Creates two different topics, two non-durable topic subscribers and one durable subscriber. Different messages are sent in each topic while subscriptions are deleted and created again.
7. **JDNI**Test: another naming service test (JDNI). Different objects are bound to the naming service and then searches are performed to test pattern matching based search.
8. **McastReceiverTest**: simple test to verify multicast is enabled on the network. Receives multicast messages sent by `McastSenderTest`.
9. **McastSenderTest**: sends multicast message that should be received by `McastReceiverTest` if multicast is enabled on the network.

6. Conclusion

To conclude, we can state that we have built an integrated approach for developing and using applications for collaboration in the MANET environment.

Collaboration environments highly rely on group communication, so most traffic will be directed to a group, i.e., using multicast delivery. By using OMCAST, a new application level multicast protocol, we reduce the overall network traffic by benefiting from the broadcast nature of the medium.

Using the basic network primitives, we have built a middleware with a rich set of communication mechanisms and membership information: named communication channels, publish/subscribe and naming services together with membership information and events.

The development of the plug-in framework allows building final applications in a rapid and simple manner. These applications, named plug-ins, utilize middleware services in order to create collaborative applications for the MANET environment.

With all these components, we believe that AGORA is a good solution for developing collaborative applications for MANETs in an easy and straightforward manner.

7. References

- [1] C.M. Cordeiro, H. Gossain, and D.P. Agrawal, "Multicast over Wireless Mobile Ad Hoc Networks: Present and Future Directions", *IEEE Network*, vol. 17, no. 1, 2003, pp. 52-59.
- [2] R. Meier and V. Cahill, "Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications," *First Workshop on Middleware for Network Eccentric and Mobile Applications (MiNEMA)*, Dublin, Ireland, 2004.
- [3] M. Musolesi, C. Mascolo, S. Hailes, "EMMA: Epidemic Messaging Middleware for Ad hoc networks". *Journal of Personal and Ubiquitous Computing*, Springer, Vol 10, No 1, February, 2006, p. 28-36.
- [4] D.Bottazzi, A.Corradi, R.Montanari, "AGAPE: A Location-Aware Group Membership Middleware for Pervasive Computing Environments", in *Proc. of the 8th International Symposium on Computers and Communications (ISCC2003)*, IEEE Press, Turkey, July 2003.
- [5] Lu Yan, Kaisa Sere, Xinrong Zhou, Jun Pang, "Towards an Integrated Architecture for Peer-to-Peer and Ad Hoc Overlay Network Applications", In *Proc. 10th Workshop on Future Trends in Distributed Computing Systems - FTDCS'04.*, pp. 312-318.
- [6] M. Bisignano, G. Di Modica, O. Tomarchio, "JMobiPeer: a middleware for mobile peer-to-peer computing in MANETs". *First International Workshop on Mobility in Peer-to-Peer Systems (MPPS) (ICDCSW'05)* pp. 785-791.d
- [7] Alf Inge Wang, Tommy Bjornsgard and Kim Saxlund, "Peer2Me - Rapid Application Framework for Mobile Peer-to-Peer Applications", *International Symposium on Collaborative Technologies and Systems (CTS 2007)*, Orlando, Florida, USA, May 2007.
- [8] J. Xie, et al., "AMRoute: ad hoc multicast routing protocol", *ACM/Baltzer Mobile Networks and Applications*, special issue on Multipoint Communications in Wireless Mobile Networks 7 (6) (2002).
- [9] M. Ge, S. V. Krishnamurthy and M. Faloutsos. "Application versus network layer multicasting in ad hoc networks: the ALMA routing protocol", *Ad Hoc Networks* Volume 4, Issue 2, March 2006, 283-300.
- [10] M. A. Kaafar, C. Mrabet, T. Turletti, "A Topology-Aware Overlay Multicast Approach for Mobile Ad-Hoc Networks", *AINTEC*, Bangkok, Thailand, November, 2006.
- [11] S. Blödt, "Efficient End System Multicast for Mobile Ad Hoc Networks", in *Proc. of PERCOMW 2004*, Orlando, Florida, USA, March 2004.
- [12] C. Gui and P. Mohapatra, "Efficient Overlay Multicast for Mobile Ad Hoc Networks," in *Proc. IEEE Wireless Comm. and Networking Conf.*, vol. 2, IEEE Press, 2003, pp. 1118-1123.
- [13] Gerard París, Marcel Arrufat, Pedro García López, Marc Sánchez Artigas: An Application Layer Multicast for Collaborative Scenarios: The OMCAST Protocol. *ICN 2008*: 99-104

8. Glossary

Acronyms and terms	Meaning
MANET	Mobile Ad-hoc Network
Member	A host which is currently that has joined a group in the system
Node	Machine, usually a laptop, taking part in a mobile ad-hoc network.
ALM	Application Level Multicast
DYMO	DYnamic Multi-hop On-demand protocol
OLSR	Optimized Linked State Routing

Annex 1. Related Publications