
	<p style="text-align: center;">MOBILEMAN</p> <p style="text-align: center;">IST-2001-38113</p> <p style="text-align: center;">Mobile Metropolitan Ad hoc Networks</p>
---	--

<p>MOBILEMAN</p> <p>MobileMAN Functionalities – Final Set</p> <p>Deliverable D14</p> <p>Contractual Preparation Date: August 2005 Actual Date of Delivery: 1 September 2005 Estimated Person Months: 25 Number of pages: 35</p> <p>Contributing Partners: Consiglio Nazionale delle Ricerche (Italy), University of Cambridge (UK), Eurecom Institut (France), Helsinki University of Technology (Finland), Netikos (Italy)</p> <p>Authors: Marco Conti, Emilio Ancillotti, Eleonora Borgia, Raffaele Bruno, Franca Delmastro, Enrico Gregori, Giovanni Mainetto, Antonio Pinizzotto (CNR), Jose Costa-Requena, Jarrod Creado, Mohhamad Ayyash (HUT), Jon Crowcroft, Andrea Passarella (Cambridge), Refik Molva, Claudio Lavecchia, Pietro Michiardi (Eurecom), Piergiorgio Cremonese, Veronica Vanni (Netikos)</p>
--

<p>Abstract: The aim of this deliverable is to provide the software that implements, on the Linux operating system, the functions required to set up a MobileMAN. Specifically, in addition to the revised version of the software modules already delivered in D11 -- i) CORE watchdog mechanism, ii) ad hoc routing framework, iii) p2p Pastry platform, and iv) VoIP and whiteboard applications -- we now also deliver: the CrossROAD and XL-plugin modules (which implement a subset of the MobileMAN cross-layer architecture), and the UDDI4m modules. The software modules are contained in the CD ROM associated with this deliverable and are also made available in the Software web site http://keskus.hut.fi/tutkimus/MobileMan. In addition, we deliver our <i>Ad Hoc Proxy ARP daemon (AHPAd)</i> that enables the interconnection of MobileMAN ad-hoc islands with the Internet. The <i>AHPAd</i> code is reported in the Appendix.</p>
--

	<p>Project funded by the European Community under the “Information Society Technologies” Programme (1998-2002)</p>
---	---

SUMMARY

The aim of this deliverable is to provide the software modules that implements, on the Linux operating system, the functions required to set up a campus-wide MobileMAN, as identified in D5, D10 and D13. Indeed, by integrating the software modules we developed with existing code, we obtained a set of software architectures that enable us to test MobileMAN concepts and ideas. Specifically, we have: i) a legacy (layered) TCP/IP architecture on which we run a VoIP application; ii) a legacy (layered) p2p architecture on which we run Whiteboard and UDDI4m applications, and iii) a p2p cross-layer architecture on which we run both Whiteboard and UDDI4m applications. In this document we describe the main characteristics of the software modules we deliver. More details on our solutions can be found in Deliverables D5, D10 and D13. The software we are delivering is contained in the CD ROM associated with this deliverable and which is also made available in the Software web site <http://keskus.hut.fi/tutkimus/MobileMan>.

In addition, in this deliverable, we also present our solution to interconnect MobileMAN ad hoc islands among themselves and with the Internet. To this end, we implemented an *Ad Hoc Proxy ARP daemon (AHPAd)*. The *AHPAd* code is reported in the deliverable's appendix.

CONTENTS LIST

1.	INTRODUCTION.....	4
2.	SOFTWARE	6
2.1	Cooperation enforcement mechanism: Watchdog	7
2.1.1	Architecture.....	7
2.1.2	Functional Description.....	8
2.2	Routing Protocols	9
2.2.1	Common modules	10
2.2.2	Reactive modules	11
2.2.3	Proactive modules	11
2.2.4	Hybrid modules.....	11
2.3	Middleware Platforms	12
2.3.1	FreePastry.....	12
2.3.2	CrossROAD and XL-plugin.....	13
2.4	Applications.....	15
2.4.1	Whiteboard.....	15
2.4.2	UDDI4m.....	18
2.4.3	VoIP	20
3.	SCENARIOS.....	21
4.	INTERCONNECTION TO THE INTERNET.....	23
4.1	Our solution.....	24
5.	REFERENCES.....	27
6.	APPENDIX: AD HOC PROXY ARP DAEMON SOFTWARE	29

1. INTRODUCTION

One of the MobileMAN project novelties is to tackle the lack of *Integration, Experimentation and Implementations/Testbeds* in the MANET research. Specifically, the project aims to the development and validation of solutions for the relevant technical issues of self-organizing networks (routing and medium access control protocols, power management, security, and location). Whenever possible, real testbeds are (and have been) used to validate the solutions we devised to implement a MobileMAN.

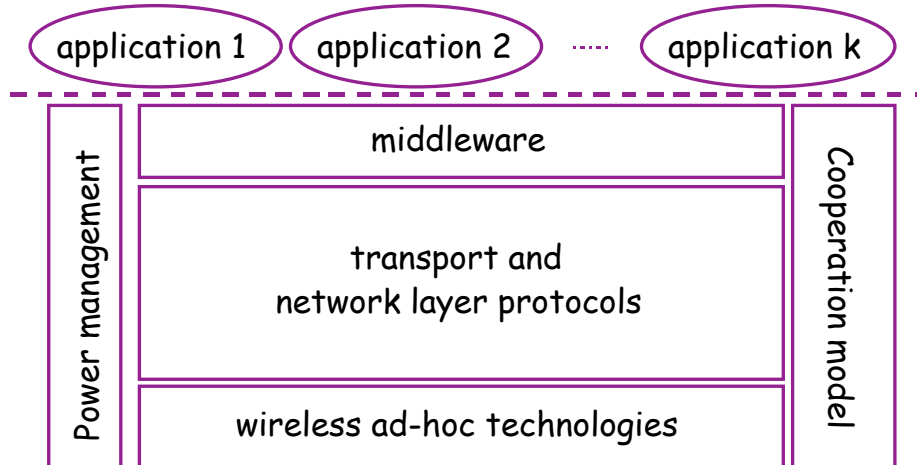


Figure 1: MobileMAN original Reference Model

In Annex 1 we identified the legacy layered architecture shown in Figure 1, as the MobileMAN reference model.

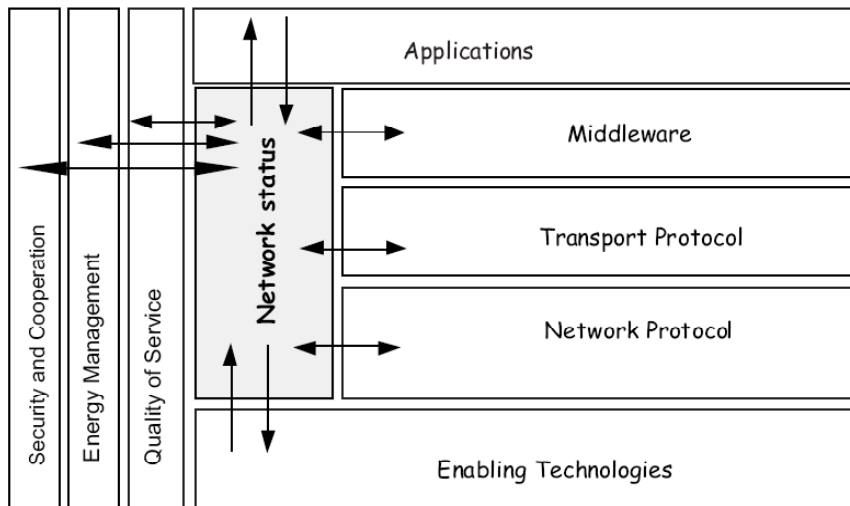


Figure 2: MobileMAN cross-layering Reference Model

According to the ideas and concepts developed during the first year of the project (see Deliverables D4 and D5), and further refined during the second year of the project (see Deliverable D10), we enhanced the reference model of MobileMAN in order to integrate the new view of “cross-layering”. The enhanced reference architecture is shown in Figure 2. In this architecture, cross layering is implemented through data sharing. As shown in the figure, the innovation of the architecture is a shared memory, “Network Status” (NeSt) in the figure, which is a repository of all

the network status information collected by the network protocols. All protocols can access this memory to write the information to share with the other protocols, and to read information produced/collected from the other protocols. This avoids duplicating the layers' efforts for collecting network-status information, thus leading to a more efficient system design. In addition, inter-layer co-operations can be easily implemented by variables sharing. However, protocols are still implemented inside each layer, as in the traditional layered reference architecture.

In constructing real testbeds, whenever possible, we exploited software modules already available in the literature, while we concentrated our software developments on novel and unsolved issues. Hereafter, with reference to the MobileMAN architecture (both legacy and cross-layer), we present where the project software development efforts have been concentrated.

2. SOFTWARE

Hereafter we (briefly) introduce the software modules we have implemented for constructing a MobileMAN network according to both the legacy and the enhanced (cross-layer) architecture.

To maximize the usefulness of project results, we limited the software development to novel mechanisms and protocols we identified during the project, while we exploited as much as possible valuable concepts/solutions (and the corresponding software implementations) already available in the literature.

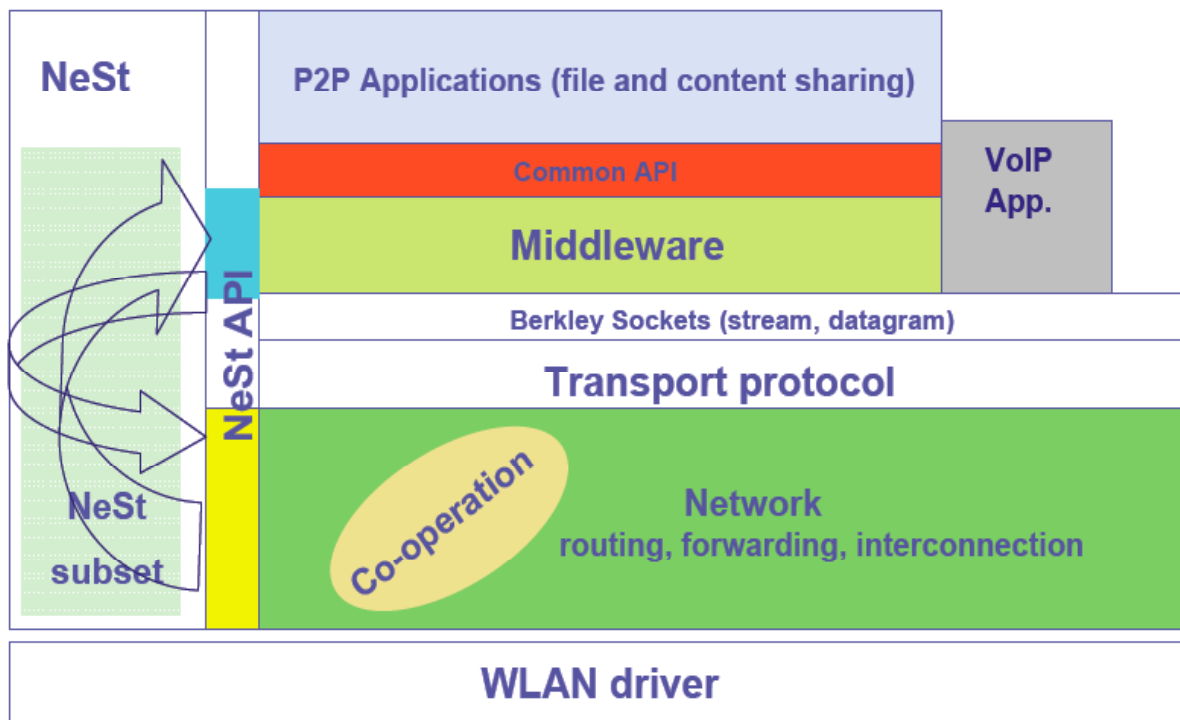


Figure 3: MobileMAN Software Architecture

Specifically, with reference to the MobileMAN software architecture shown in Figure 3, we concentrated our development efforts on colorful boxes while white boxes represent the existing modules that we integrated in our software architecture.

As far as the legacy architecture, we focused mainly on integration (and porting) of off-the-shelf HW and SW components (802.11, AODV and OLSR for routing, Pastry as middleware platform, etc.) in a MobileMAN testbed. In addition, when available solutions were not suitable for the project aims, we implemented our own HW/SW solutions (e.g., 802.11 enhanced card, applications).

For the cross-layer architecture, we implemented a subset of it to be used as a proof of concept of the cross-layer principle. Specifically, we have designed and implemented:

- i. a revised version of Pastry middleware (named CrossRoad) to optimize its performance by exploiting cross layer interactions;
- ii. a minimal set of the NeSt functionalities to guarantee cross layer interactions between the middleware (CrossRoad) and the network layer;
- iii. an enhanced version of the Common API in order to support p2p applications (whiteboard and UDDI) on top of the cross layer architecture.

Hereafter, we will present the software modules used to construct the MobileMAN software architecture. By integrating existing software modules with those implemented by us, we have obtained a set a testbeds, see Section 3, that enable us to test MobileMAN concepts and ideas.

2.1 Cooperation enforcement mechanism: Watchdog

This section describes the watchdog module as part of the implementation of reputation-based cooperation enforcement mechanism for mobile ad hoc networks (i.e. CORE).

2.1.1 Architecture

The cooperation enforcement mechanism proposed for the MobileMan architecture is the CORE mechanism. CORE is a collaborative monitoring mechanism based on reputation that strongly binds network utilization to the correct participation to basic networking function like routing and packet forwarding. CORE is being implemented as a Linux user-space daemon that runs on all the nodes of a MANET and can possibly be used with different routing protocols. In the future CORE will be able to store reputation information in a local storage accessible from different layers of the MobileMAN stack in order to help inter-layer optimization. CORE can be decomposed in three building blocks as shown in the CORE mechanism architecture depicted in *Figure 4*:

- 1) A monitoring mechanism implemented as a MAC layer sniffer. It monitors the packets that pass across layer 2 of the TCP/IP stack of a node and deduces whether neighbors are participating or not to basic networking functions. Monitoring of neighbors behavior is achieved by setting the WLAN card in promiscuous mode.
- 2) A reputation function that according to the output of the MAC layer sniffer calculates a reputation value for each neighboring node and marks neighbors as selfish when their reputation falls below a given threshold.
- 3) A punishment mechanism that punishes neighbors marked as selfish. According to a simple punishment model, a node punishes a selfish neighbor by refusing the forwarding of the selfish neighbor packets. A selfish node can be reintegrated in the MANET if it restarts performing packet forwarding function.

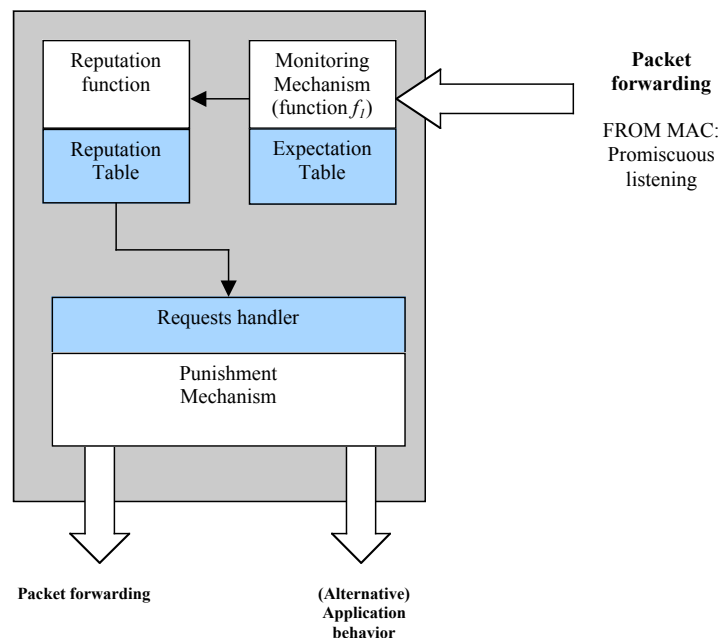


Figure 4 CORE architecture.

In the earlier stages of CORE design and implementation we consider the reputation as depending only on the participation to the packet forwarding function and concentrate on selfish nodes detection.

Further research and development will include integration of participation to the routing function as an input of the reputation function as well as design of the punishment mechanism.

Detailed description of the CORE implementation concepts such as the expectation table and the reputation function can be found in section 4.5 of MobileMAN deliverable D10.

2.1.2 Functional Description

CORE watchdog is implemented as a Linux daemon and packaged for installation on PDAs running Familiar Linux distribution.

Once it is installed following the instructions contained in the README file, it can be run simply by typing “watchdog” on the console.

CORE watchdog needs to be executed in cooperation with a WLAN card that works in promiscuous mode. It has been successfully tested on the Dell TrueMobile 1150 WLAN card.

CORE watchdog module monitors neighboring nodes behavior and writes output to the console. Some attributes of each packet captured within the transmission range are shown to the user, such as source and destination IP and MAC addresses, TCP sequence number and so on.

CORE uses MAC addresses to identify neighbors. Each time CORE watchdog detects a selfish behavior (i.e. a neighbor does not forward a packet coming from the node where the CORE watchdog is running), a line with the information about the selfish neighbor is printed to the console. Similar output is printed to the console when cooperative neighbors are detected.

The detection of selfish behavior is made by overhearing the packets that neighbors forward. When a watchdog waits for a neighbor to forward a packet, it temporarily stores the packet into the expectation table.

Operations on expectation table contents are printed on the standard output as well as other messages that help the user to understand the operations of the CORE watchdog.

2.2 Routing Protocols

The Ad Hoc routing framework is a software package, which can support different Ad Hoc networks routing protocols, such as proactive, reactive and also some hybrid solutions. The Ad Hoc routing framework can be installed in a node (PDA or Laptop) that runs Linux Operating System. With this framework, we could add new routing protocols and other functionalities, such as naming and service discovery. This section describes the existing components of the Ad Hoc framework and all the subsystems including interfaces and the basic functionalities implemented in the framework.

The framework has been designed in separate components with clearly defined interfaces. This allows an easy integration of these components and the possibility of adding new functionalities. The benefit of this component-based or plug-in design is that each individual component could change its internal implementation while all components can still be integrated together if the interfaces are kept consistent. This allows having a complex system, which can be divided into small modules that are easy to implement.

The framework provides general functionalities for both proactive and reactive routing protocols. The existing framework includes a reactive protocol (e.g. AODV [1]) and a proactive protocol (e.g. OLSR [2]). In order to test the framework implementation with constrained devices, in addition to laptops the framework is integrated into a small number of Personal Digital Assistants (PDA) nodes (iPAQ). The iPAQ is a mobile node where the original operating system is PocketPC 2002. The Operating system has to be changed to Linux in order to integrate the framework. Linux supports the ARM architecture, which is used in the iPAQ.

We use the Familiar [3] operating system, which is a Linux portable version used on iPAQ. It is a tailored Linux version to fit into limited resources of mobile devices. Because of the limited space for the file system, we cannot install the Linux kernel source and a compile tool chain into iPAQ. This means we cannot build native executables on iPAQ directly. In addition, the default Familiar kernel does not include Netfilter (an operating system services for manipulating IP packets), but we need it for the framework. For these reasons, we have to compile everything in a Linux [4] PC to make ARM executables. Then, we transfer them to iPAQ and run them. This can be achieved by using a GNU/gcc [5] compile tool chain for ARM.

The Ad Hoc Framework consists of four subsystems: the Common Cache Registry Server, the Reactive modules, the Proactive modules and the Hybrid modules. The Common Cache includes all the modules that must be kept constantly running in the node since they store routing information and other data used by the other modules. The Reactive modules consist of the software modules that implement the reactive routing protocols (e.g. in this case the Reactive module consists of the module that implements AODV). The proactive modules consist of the software modules that implement proactive routing protocols (e.g. in the actual framework the proactive modules contain only a software module that implements OLSR). Finally, the hybrid modules include all modules that will implement hybrid routing protocol such as ZRP. The four modules of the Ad Hoc framework consist of independent software components that implement specific routing protocols and store routing information into a single cache. The Ad Hoc framework architecture is shown in Figure 5.

- Common modules: Common Cache Registry Server, Common Cache and Registry.
- Reactive routing modules: AODV module.
- Proactive routing modules: OLSR module.
- Hybrid routing modules.

2.2.1 Common modules

The common modules consist of a Registry, the Common Cache and the Common Cache Registry Server that communicates with the independent routing modules. The Common Cache keeps routing and other information collected by the routing protocols that are running simultaneously in the node. The Registry stores information of the routing protocols running in the node.

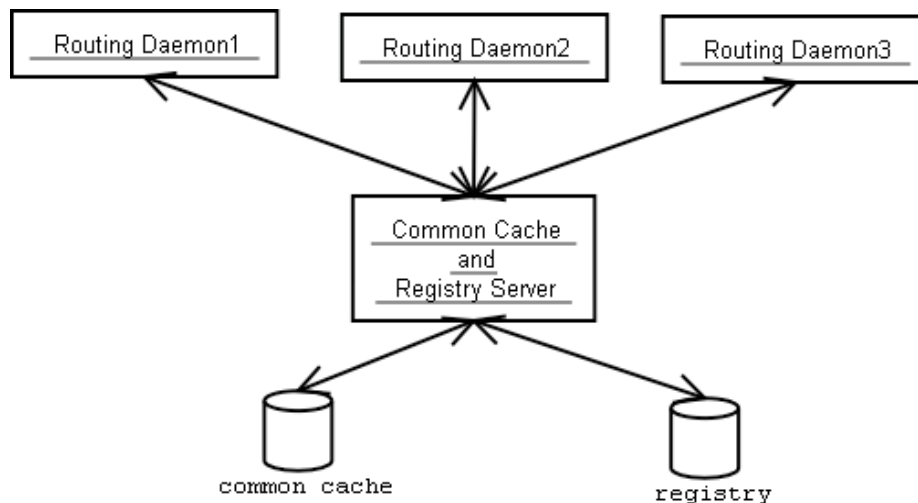


Figure 5 Ad Hoc framework architecture

The aim of the Ad Hoc framework design is to contain several independent routing protocols running simultaneously as daemons (the actual implementation of the framework contains two routing daemons: AODV and OLSR). The Common Cache and Registry Server (CCRS) act as the front end of two data repositories: the Common Cache and the Registry. The daemons of the routing protocols act as clients to the Common Cache and Registry Server for accessing the Common Cache and the Registry.

The Common cache stores all the route information that the CCRS receives from all routing protocols running in the node as daemons. The Registry contains the state information of all routing protocols daemons (e.g. active, inactive, etc). The state information consists of configuration parameters of the protocol that is active but also other parameters for sending messages to the daemon in order to change its configuration during runtime. This allows the implementation of new routing algorithms by exploiting the information provided by existing routing protocols that are already running (e.g. hybrid routing protocols such as ZRP could be implemented using existing AODV and OLSR daemons).

The Common Cache Registry Server (CCRS) is one of the most important common modules in the Ad Hoc framework. The CCRS keeps listening to specific messages coming from the routing protocols daemons that want to communicate either with the Common Cache or with the Registry.

Thus, when one routing protocol daemon starts running, it must upload its state information and other protocol parameters into the registry through CCRS. Moreover, during the lifetime of the routing protocol it updates its own routing table and the routing information in the common cache through the CCRS. Thus, the Common Cache always stores the routes discovered by the separated protocols running in the device simultaneously. Meanwhile, the registry keeps the latest state information of the routing protocol daemons running in the node.

When one routing daemon wants to know the status of other daemons, it can ask the information to CCRS. Different daemons can also communicate through CCRS.

The communication between the daemons and the CCRS is implemented by different messages defined in request/reply format. The CCRS server and the routing protocol acting as clients are running in different processes and they communicate through a well-known port.

2.2.2 Reactive modules

The reactive modules consist of the software components that implement reactive routing protocols (e.g. AODV).

AODV Module

This section describes a reactive module that implements the specific AODV reactive routing protocol. We select AODV because it is the most widely implemented reactive algorithm.

The AODV module is based on UU-AODV [6]. The actual implementation of the Ad Hoc framework contains version 0.8 of the UU-AODV. The aim of the Ad Hoc framework is not to re-implement existing protocols but integrate existing implementations and reuse as much as possible existing implementations that are already tested and debugged into a single framework. The existing UU-AODV implementation has to include an additional component to communicate with the CCRS implemented as part of the framework. Thus, the UU-AODV is included into the framework as independent routing module that only provides the AODV protocol logic. An additional component named CCRS proxy is added within the UU-AODV to slightly modify the AODV behavior. The CCRS proxy will store the routes discovered by AODV into the CCRS so making the routes available to the Linux kernel routing table for the node communications.

2.2.3 Proactive modules

The proactive modules consist of the software components that implement proactive routing protocols (e.g. OLSR).

OLSR Module

OLSR is a proactive routing protocol that periodically sends control messages to maintain the knowledge of the network topology. OLSR protocol is a link state protocol where a selected set of nodes broadcast over the network the list of its neighbors. In this case all the nodes know all the others. Therefore, the nodes have all the routes and thus the shortest path to all the destinations.

The OLSR module included in the Ad Hoc framework follows the same approach as the AODV module. Thus, the OLSR module is based on UNIK-OLSR [7]. The actual implementation of the Ad Hoc framework contains version 0.4.7 of the UNIK-OLSR. Similarly to the AODV, the existing UNIK-OLSR implementation has to include an additional component allowing the communication between the UNIK-OLSR and CCRS. Thus, an additional component named CCRS proxy is added within the UNIK-OLSR to slightly modify the UNIK-OLSR behavior. The CCRS proxy will store the routes discovered by OLSR into the CCRS so making the routes available to the Linux kernel routing table for the node communications.

2.2.4 Hybrid modules

The hybrid modules consist of the software components that implement hybrid routing protocols (e.g. ZRP) but the actual Ad Hoc framework implementation does not contain any implementation.

2.3 Middleware Platforms

To implement the legacy architecture we integrated in the MobileMAN software architecture a p2p middleware platform based on Pastry (*FreePastry*) that enables us to run simple p2p testing applications on top of the multi-hop ad hoc network. FreePastry is briefly described in Section 2.3.1.

To optimize the middleware-platform performance by exploiting cross layer interactions, we have designed and implemented an enhanced version of Pastry middleware, named CrossRoad. CrossRoad and the part of the Network Status required to exploit cross-layer interactions between routing and middleware are described in Section 2.3.2.

2.3.1 FreePastry

FreePastry [8] is an open source implementation of the Pastry overlay network model. Free Pastry is developed by the Rice University and it completely follows Pastry principles as described in [9]. Particularly it defines the internal data structures aimed at maintaining overlay's information, it defines the bootstrap node needed to establish and join the ring and related remote connections needed to recover those information.

In order to test FreePastry functionalities, we defined a simple application of *Distributed Messaging*. This service is an example of messages' exchange between peers, storing contents on distributed nodes of the network. Each node defines a "MailTable" data structure containing a variable number of records. These records are represented by (mailboxID, msgList) pairs. Each node can create more than one mailbox, specifying a unique identifier for each of them. They are created on demand through a "create" request message, sent on the overlay specifying the mailboxID as the key value. In this way a single node can maintain the mailbox of different nodes, depending on the distribution of the identifiers on the ring and they can send messages with the destination's mailboxID as routing key. The identifier of the first node is represented by the hash function applied to the IP address of the local node, while the identifier of the others nodes is autonomously calculated by FreePastry specifying only the IP address of a physical neighbor already present in the overlay. At the same time the identifier of each mailbox is represented by the hash function applied to a string chosen by the user as the identifier of the mailbox (mailboxID). Each node maintains messages belonging to mailboxes with ID logically closest to its logical address. In this way, specifying the mailboxID as the research key of the routing message, the subject-based routing is correctly implemented and mailboxes are uniformly distributed on the nodes joining the ring.

Once a mailbox is created and some messages received, a node can send a "get" request for this mailbox to download messages without deleting the related entry in the Mailtable, except the case in which a "delete" request is forwarded. When the node storing the mailbox receives the request, it directly sends the message list to the requiring node, forcing the Pastry routing protocol to a single peer-to-peer connection.

We can summarize as follows basic steps for the definition and the start up of the Distributed Messaging application. First of all, the application requires the user to specify if the starting node is the first node of the ring or not:

- 1) if the node is the first node that participates to the service, Pastry defines the ring starting from the IP address of the local node
- 2) if the requiring node has a knowledge of the ring, it has to specify the IP address of a known node (physical neighbour). In this case Pastry automatically recovers the ID of the specified node and initializes the routing tables of the new node directly connecting with other participants of the overlay.

A preliminary phase to test original FreePastry functionalities was conducted using a set of PC connected to a LAN, forming a classical P2P system. After an exhaustive test of FreePastry implementation on the wired network, we configured the application for an ad hoc scenario using a group of laptops equipped with PCMCIA wireless cards [10]. After setting up the connection parameters for ad hoc mode and synchronizing nodes on the same frequency, it is necessary to explicitly set the correspondence between the hostname and the ad hoc IP address before running the application. At this point, to start the application and join the ring, is sufficient to use the local IP address if the node is the first of the ring, or the IP address of a known host in the range of the requiring node.

2.3.2 CrossROAD and XL-plugin

CrossROAD [11] represents an optimized p2p system for ad hoc networks, based on the Pastry overlay network model [9]. Specifically it exploits a cross-layer architecture, using network routing table information in order to maintain a correspondence between the physical network topology and the logical address space, where nodes and data are mapped. In this case the cross-layer interaction is limited to the middleware and the network layers. In fact, in order to have a complete and updated knowledge of the network topology, a proactive routing protocol is needed, and for this reason we selected an open source implementation of OLSR (Unik-OLSR v.0.4.8 [7]), that we had already tested from the functionality and overhead standpoints, as described in Deliverable D8 [10]. In addition this implementation allows the definition of libraries dynamically loaded by the routing daemon at the startup, in order to export routing information to other applications, or to define additional information to be sent on the network through the proactive flooding of routing packets. These libraries are called plugins.

In case of a structured p2p system, each overlay consists of all nodes providing the same service, but in the Pastry model each node has only a partial knowledge of the overlay due to the limited dimensions of its internal data structures, and a lot of remote connections are needed to initialize and maintain them. Instead, in case of CrossROAD, every node joining the overlay can directly know all the others that are providing the same service, exploiting a new Service Discovery protocol that associates a unique identifier to each service, and broadcasts this information on the network, piggybacked in routing packets.

In this way, a plugin, called XL-plugin, has been defined in order to encapsulate additional information in routing packets. This information is represented by services identifiers, used to associate to each node the list of services locally provided. When OLSR receives a routing message containing this additional information, it passes the contents to XL-plugin that provides to store services identifiers of other nodes in its local data structures. For this reason XL-plugin maintains two local data structures: LocalService Table and GlobalService Table. Specifically, the LocalService Table maintains the list of services provided by the local node, while the GlobalService Table maintains, for each service present in the network and currently running on CrossROAD, the list of nodes providing it. All entries are timed out in order to preserve the consistency of the service information. In this way, when a node starts running an application on top of CrossROAD, it declares its service identifier and CrossROAD directly establishes a local connection to the plugin in order to receive the list of nodes taking part to that specific overlay. Then, when the local application sends a message with a specified key value, CrossROAD first checks the consistency of its internal data structures with the list provided by the plugin, then it determines the best destination for that key and directly sends the message to it. More details on software architecture of CrossROAD and XL-plugin can be found in deliverable D13 [12].

In order to test CrossROAD and XL-plugin functions, comparing them with results obtained from FreePastry [8] experimentation, the same Distributed Messaging application has been used.

In addition other two applications have been developed on top of CrossROAD: Whiteboard from University of Cambridge [13] [14] and UDDI4m from NetiKos.

2.4 Applications

As explained in Deliverables D10 and D13, we identified a set of applications that enable us to test both the legacy and enhanced architecture. Specifically, taking also into consideration users indications we included in the MobileMAN software architecture:

- co-operative tools for document/content sharing based on a P2P architecture (Whiteboard and UDDI);
- Voice over IP applications exploiting the legacy TCP/IP protocol stack.

2.4.1 Whiteboard

This section describes the functionality of the Whiteboard as a peer-to-peer multicast application suitable for Ad Hoc networks. The Whiteboard developed by the Cambridge University is a Peer-to-Peer multicast application based on any structured overlay network implementing the commonAPI interface [18]. On top of it, the Whiteboard requires a p2p multicast layer to handle group membership and data delivery. In order to make the design as flexible as possible, these functions were implemented independently. We have developed the Whiteboard application for both reference network architectures used as reference within the MobileMAN project, i.e., the Legacy and the Cross-Layer Architecture (see Deliverable D13). Finally, it is worth noticing that the Whiteboard application should be looked at as a placeholder for several Group-Communication Applications (instant messaging, file sharing, video streaming, etc) that could be deployed on ad hoc networks based on the same protocol stacks.

In this section we describe the main functionalities of the Whiteboard application and the p2p multicast substrate. Then, we focus on how the software we have developed is integrated into the reference network architectures.

2.4.1.1 Whiteboard Functionalities

Target users of the Whiteboard application create a virtual group (a community) for a limited amount of time in order to exchange dynamically generated content (e.g., drawings and text). From this standpoint, the Whiteboard shares similarities with Instant Messaging applications. Figure 6 shows the graphical interface presented to the user once the Whiteboard is started.

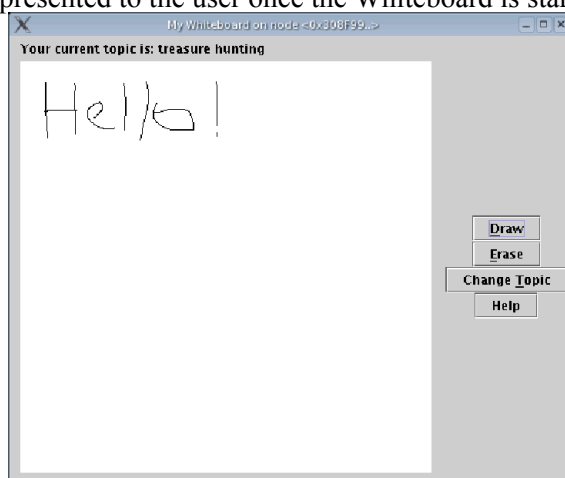


Figure 6. The Whiteboard interface

The Whiteboard functionalities are very simple and intuitive. Firstly, a user has to choose the *topic* he wants to associate to. All users subscribing to the same topic are allowed to exchange data through the Whiteboard. Then, the user can draw strokes on the Whiteboard canvas. Locally generated strokes are propagated to all users subscribed to the same topic. On the other hand, the Whiteboard running on the user's device renders on the canvas strokes generated by other users. Finally, the user can erase the canvas and change the topic he is subscribed to.

The graphical input and output is handled through the `awt` and `swing` Java packages. Specifically, an event handler is triggered upon mouse gestures inside the canvas areas. The event handler draws the strokes on the canvas, and invokes Scribe to deliver the new strokes to the other group members. On the other hand, the callback function invoked by Scribe when receiving new messages from other peers draws the strokes contained in the message on the local canvas.

2.4.1.2 P2P Multicast Substrate Functionalities

The Whiteboard Application has been developed by exploiting middleware services provided by standard p2p systems. Specifically, it exploits a p2p structured overlay network such as Pastry and CrossROAD (i.e., implementing the commonAPI [18]). Furthermore, it also exploits Scribe, a p2p multicast algorithm built on top of such an overlay network [19]. Since the Pastry and CrossROAD functionalities have been already described in this deliverable, we now briefly sketch the main Scribe functionalities.

Scribe exploits Pastry-like routing to build multicast groups. From the standpoint of the application running on Scribe, the group is identified by a topic. Scribe uses the hash function provided by Pastry (or CrossROAD) to generate the topic id (tid) in the logical space of node ids. In order to join the Scribe tree, nodes send a join message on the overlay with key equal to tid. This message reaches the next hop (say, N) towards the destination on the overlay network. The node originating the join message is enrolled as a child of N. If not already in the tree, N itself joins the tree by generating a join message anew. Eventually, such a message reaches the node whose id is the closest one to tid and is not propagated further. This node is defined as the root of the Scribe tree. Application messages are sent on the overlay with key equal to tid. Hence, they reach the Scribe root, which is in charge of delivering them over the tree. To this end, it forwards the messages to its children, which further forward them to their children, and so on. Finally, the Scribe maintenance procedure is as follows. Each parent periodically sends a HeartBeat message to each child (application-level messages are used as implicit HeartBeats). If a child does not receive any message from the parent for a given time interval (20 s in the default case), it assumes that the parent has given up, and re-executes the join procedure. This simple procedure allows node to discover parent failures, and re-join the tree, if the case.

The Scribe services are exploited by Whiteboard in a quite straightforward way. When a user selects a topic to associate to, Whiteboard invokes the Scribe `join()` procedure for that topic. Scribe generates and propagates a join message as explained above. When a new stroke is drawn on the canvas, the Whiteboard invokes the Scribe `multicast()` function, which triggers the message delivery procedure described above. Finally, the Whiteboard is notified by Scribe when a new message (generated by some other user) arrives. This is achieved through Whiteboard callback functions that are registered upon associating to the Scribe substrate, and are invoked by Scribe itself.

2.4.1.3 Whiteboard and Scribe in the reference network architectures

The Whiteboard application and the Scribe substrate have been integrated in both network architectures used in the framework of the project. A pictorial representation is given in Figure 7. As far as the Legacy architecture, the Scribe integration has been straightforward, since the

FreePastry package [8] already contains a Scribe implementation to be run on top of the standard Pastry. Furthermore, the Whiteboard application has been ported to the Pastry platform, since it was originally designed for Bamboo [20]. Then, the Scribe software has been ported to CrossROAD. This was sufficient to achieve a complete porting to the Cross-Layer architecture, since we took care of Scribe to maintain the same interface towards upper layers.

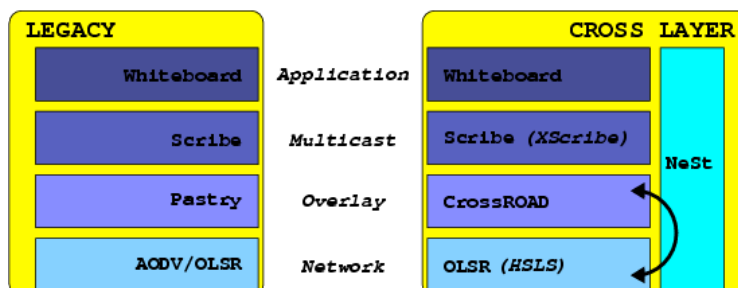


Figure 7. Whiteboard and Scribe in the reference network architectures.

2.4.1.4 Software Packages

Both versions of Whiteboard and Scribe have been implemented in Java. To simplify the installation and running of Whiteboard, we have generated two self-contained jar packages containing all the Java software required to run the Whiteboard in the legacy and in the Cross-Layer Architecture, respectively. The routing protocols' code has not been included in the packages, and should be downloaded and installed separately. The version of OLSR enhanced with XL-plugin is needed for the cross-layer solution.

Both packages can be run on any Java Virtual Machine, version 1.4.2 and above. However, all the tests we have done have been carried out on Linux platforms. The two packages bundled with this deliverable are named `wb_pastry_v1.0.tar.gz` and `wb_cr_v1.0.tar.gz`, respectively.

2.4.2 UDDI4m

This section describes the functionality of the UDDI4m service and the application that uses this service. The UDDI4m (*UDDI for manets*) [15] service is a publishing and discovering service for mobile ad hoc network. This definition exploits the traditional UDDI standard introducing updates to fit into the ad hoc environment, considering the features of this type of networks.

UDDI4m service

The UDDI4m is introduced as a new layer between middleware and application layer. This *service layer* provides a service location distributed among all nodes in the network that are active at given moment. The UDDI4m has been designed for peer-to-peer network where all the nodes work at the same level and they have some policies to cooperate between them. Each UDDI4m node has a server side and client side. The server side stores information about services provided by nodes of the network, while the client side is used to publish and recover information about a specific service. In order to optimize the distribution of services information on ad hoc networks, the UDDI4m exploits the presence of a structured overlay network: Pastry in the legacy architecture and CrossROAD in the cross-layer architecture. More specifically, UDDI4m defines a categorization of services provided by nodes of the network, used by the overlay as research key for the subject-based routing.

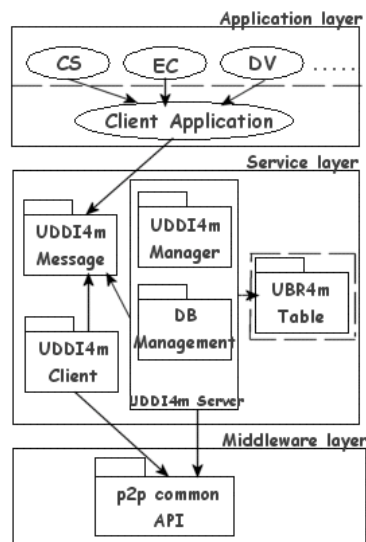


Figure 8. architecture software UDDI4m service

Figure 8 shows the software architecture of the UDDI4m service. The architecture has been implemented using a modular architecture including the following components:

- The *UDDI4m client* generates requests to the *UDDI4m server* module.
- *UDDI4m_Service* module is the core of the UDDI4m service, implementing the methods of the service: *publishService*, *findService*, *updateService* and *deleteService*;
- *DB Management* implements the data structure.
- *UBR4m Table* module manages the database.
- *UDDI4m Message* module implements the messages exchanged among nodes in the network by middleware layer.

Additional details of the UDDI4m service and the specification of the modules are included in deliverable D13 [12].

The UDDI4m service functionalities are:

- Publish a service on the distributed UBR4m registry.
- Recovery information relative to a service available in the network.
- Update or delete information relative to a service available in the network.

The UDDI4m service and all the modules are implement in java language to simplify the interaction with the lower overlay network directly implementing the P2P commonapi supported by Pastry and CrossROAD.

All modules of the UDDI4m service are packaged in a jar file called uddi4m.jar.

Client Application

The client application is a web application that uses the UDDI4m functionalities. This application provides to the users a user-friendly interface to access the functionalities provided by the UDDI4m service.

This version of the software provides the following functionalities:

- Publish a service on the distributed UBR4m registry.
- Recovery information relative to a service available in the network.

The user has to follow these steps to access the UDDI4m client application:

- Open a web browser and to connect to local url <http://localhost:8080/uddi4m/index.jsp>. The web page is opened and two choices are shown: PUBLISH or FIND service.
- The user can choose whether to publish or find a service:
 - A) In case the user wants to publish its own service, then the application provides him a html form page where the user has to insert all information related to the specific service, included its category. Then a publish message containing all data is sent to the overlay network specifying the service category as key value.
 - B) In case the user wants to find a service the application provides an html page with the list of service categories defined a priori. After the user selection, a find message is sent to the overlay with the selected key. The node with logical identifier closest to the key replies directly to the requiring nodewith the list of nodes currently providing one or more services included in that category. The reply message contains also for each node the “*access point url*” that can be used by the user to access the service on the selected node.

Functional Description

The uddi4m.jar module is implemented in java language and the client application is development in jsp (java server page). The device running the application has to satisfy the following software and hardware requirements:

The software module that represents the service layer is implemented in java language and the client application is developed as a jsp (java server page). The device running the application has to satisfy the following software and hardware requirements:

- Linux operative system
- Web server tomcat
- Mysql database to create the UBR4m registry
- OLSR routing protocol enhanced with the XL-plugin
- CrossROAD package
- Wifi card 802.11 b/g

2.4.3 VoIP

The Voice over IP is a real time application that is quite demanding for Ad Hoc networks. However, voice communications is a service that will provide added value to Ad Hoc networks since users will benefit from a communications without infrastructure support.

Following the same criteria and trying to re-use existing implementations we faced several problems when considering devices with limited resources such as PDA (i.e. iPAQ). The existing implementations of VoIP services required devices with enough processing power. Therefore, in order to develop a VoIP service for real scenarios including portable devices with low resources, we had to implement light version of VoIP application.

The VoIP application included in the Ad Hoc framework contains two main modules; signaling module and data transport module. The results show that a VoIP service can be provided in Ad Hoc networks with reasonable quality. However, since users have a good VoIP service with fixed networks the existing implementation requires a QoS module in order to enhance the service quality and meet user expectation.

2.4.3.1 Signaling module

The signaling module consists of the software component that will initiate the VoIP session with other peer nodes in the Ad Hoc network. This module has been implemented specifically for the Ad Hoc framework since existing implementations did require excessive resources (e.g. CPU, memory, etc). The signaling module implements the SIP signaling protocol and utilizes IP addresses for finding the peer nodes to initiate the VoIP session. The SIP signaling protocol can run on UDP or TCP protocol but in order to minimize the requirements for maintaining the session state in the nodes, the existing implementation uses UDP as the only transport protocol. The session initiation also requires negotiating the media parameters using SDP protocol. In order to minimize the negotiation process the signaling module uses the same codec for the VoIP session (i.e. GSM). Therefore, the signaling module is compliant with the SIP protocol but having a single codec optimizes the session set-up.

The SIP module is implemented specifically for the Ad Hoc network but the GSM codec is obtained from public source [16].

2.4.3.2 Data transport module

The data transport module consists of the software component that after the VoIP session is set up, takes care of exchanging the voice packets coded with the selected media format (i.e. in this case GSM is the only codec used in the session).

The data transport module implements a RTP client for exchanging the voice packets. The RTP client implements the functions for obtaining the audio samples from the microphone, encoding them using the selected codec (i.e. GSM) and then exchange the packets using the RTP protocol. The RTP client uses an publicly available RTP library for managing the RTP messages [17].

3. SCENARIOS

By exploiting the software we developed, and integrating it with existing software modules, we obtained three (software) architectures to be used for testing MobileMAN ideas. Specifically, we have:

- 1) a legacy TCP/IP architecture on which we run VoIP applications;
- 2) a legacy p2p architecture on which we run both Whiteboard and UDDI4m applications
- 3) a p2p cross-layer architecture on which we run both Whiteboard and UDDI4m applications

Scenario 1: VoIP on legacy TCP/IP architecture

The Voice over IP (VoIP) is a real time application that is highly demanding for ad hoc networks. The VoIP application contains two main modules; signaling module and data transport module. In the current implementation, the SIP signaling protocol uses UDP as the only transport protocol. The data transport module implements a RTP client for exchanging the voice packets.

Figure 9 summarizes the software modules used in this scenario.

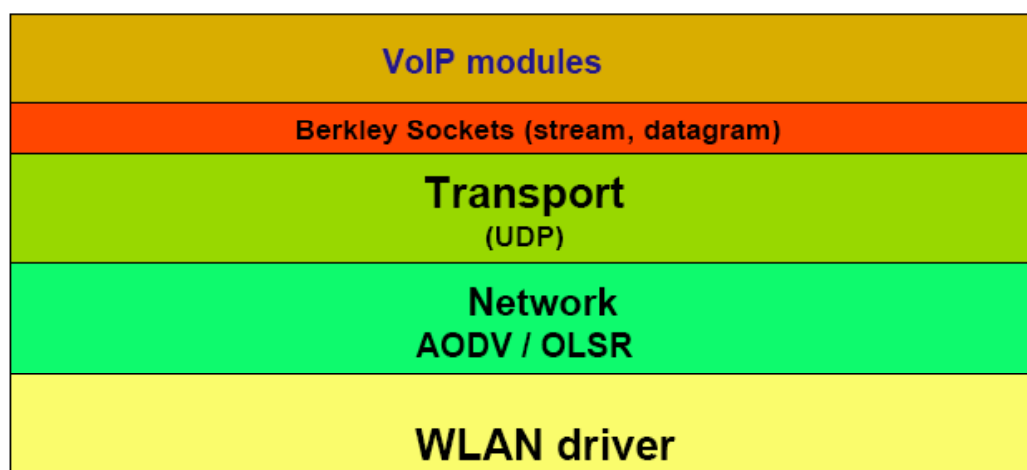


Figure 9: Scenario 1 software architecture

Scenario 2: Whiteboard and UDDI4m on legacy p2p architecture

In this scenario we investigate the behavior of content/document sharing applications based on a legacy p2p middleware architecture. Specifically, we considered a whiteboard multicast application, and a service discovery protocol and delivery mechanism, named UDDI4m, which is based on the traditional UDDI protocol. UDDI4m introduces a level between the transport and application level. The additional level of UDDI4m is composed by an overlay network, where each node may have (at the same time) the role of client and server. Service discovery is realized through the communication with the other servers on the overlay network. In this scenario the overlay network is built using the Pastry platform. Both Whiteboard and UDDI4m exploit Pastry services through the Common API.

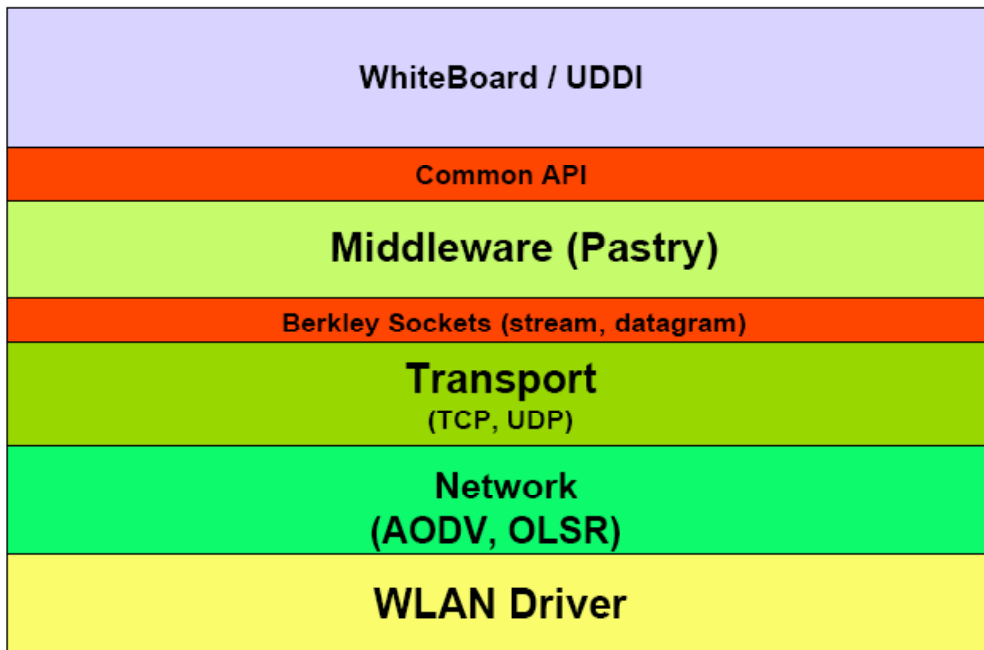


Figure 10: Scenario 2 software architecture

Scenario 3: Whiteboard and UDDI4m on cross-layer p2p architecture

In this scenario we wish to investigate the impact of cross-layer optimizations on Whiteboard and UDDI4m. To this end we implemented CrossROAD, i.e., our cross-layer version of Pastry, and the subset of the NeSt (XL-plugin) required to support cross-layer interactions between middleware and routing.

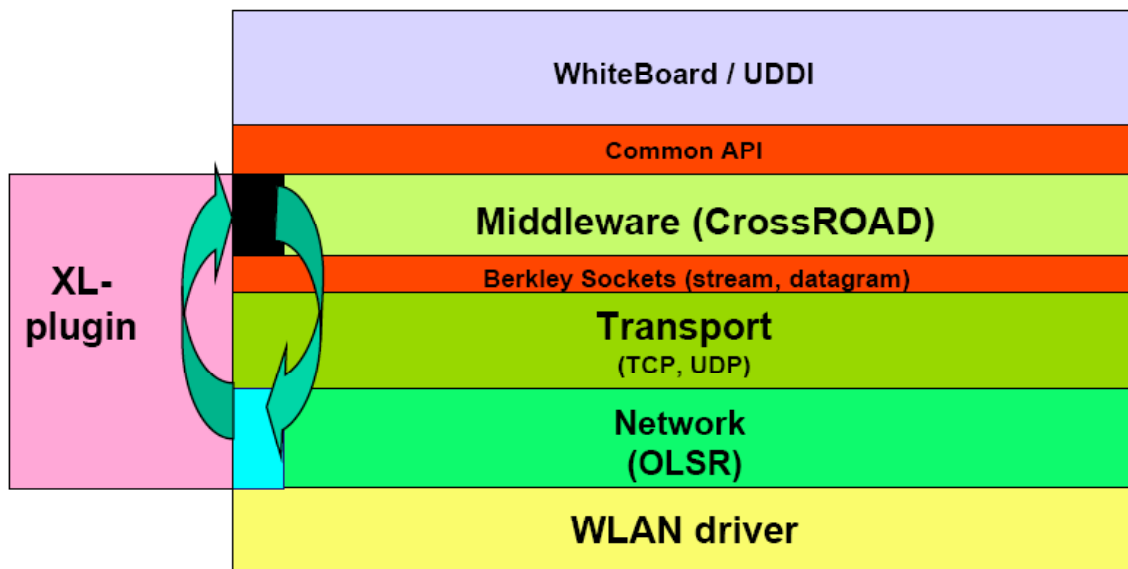


Figure 11: Scenario 3 software architecture

4. INTERCONNECTION TO THE INTERNET

In the MobileMAN we mainly focus on ad hoc networks which operate as *stand-alone* networks, i.e., as self-organized groups of nodes that operate in isolation (virtual community networks) with no connection to an external network like the Internet. However, as we discussed in Deliverable D5, an ad hoc network can either operate in isolation (virtual community network), but as shown in Figure 12, can also be interconnected to the Internet through one (or more) Internet access router, i.e., a node that has both a wireless interface to participate to the ad hoc network and a wired interface that connects it to the Internet. In this section we present an approach to interconnect ad hoc islands to the Internet and the software we developed to implement it.

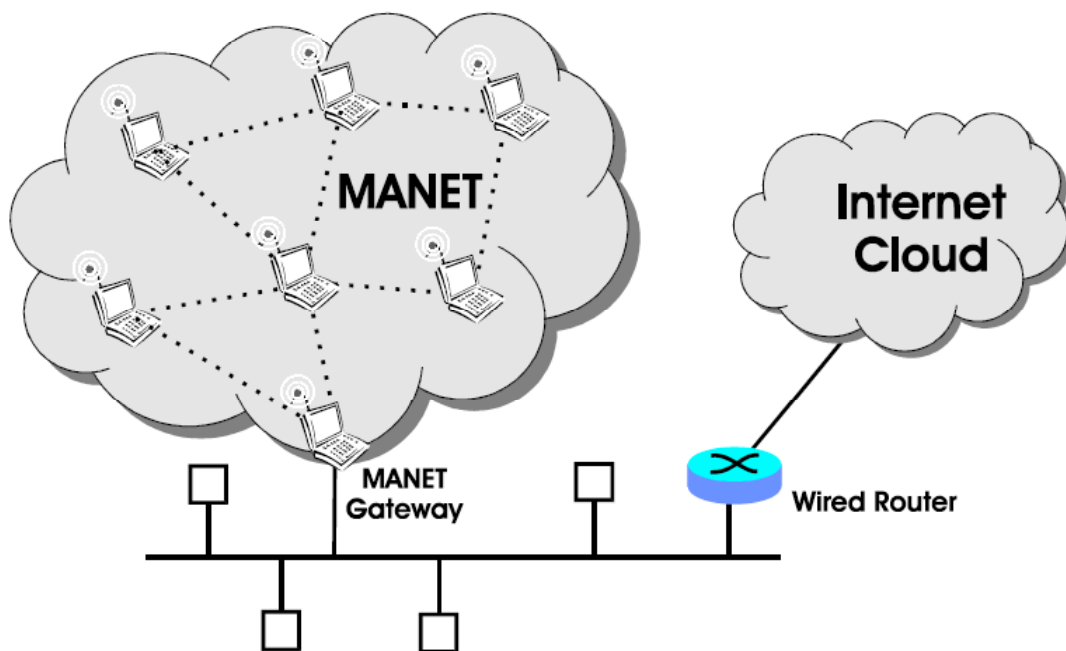


Figure 12: MANET as Internet Extension

In this section we present the implementation of a practical architecture that allows constructing a *hybrid network environment* interconnected to the Internet, in which wired and multi-hop wireless technologies are used. In particular, we envisage that group of nodes forming separated ad hoc “islands” are interconnected through wired links using special devices, hereafter indicated as gateways, which have multiple networks interfaces, both wired and wireless, such as to establish an hybrid ad hoc network. More precisely, this hybrid ad hoc network consists of: (i) mobile ad hoc nodes establishing multi-hop wireless links with other ad hoc nodes, and (ii) a fixed backbone formed by multiple gateways connected via wired links. which is interconnected with the ad hoc islands via the gateways’ wireless interfaces. This backbone is interconnected with the ad hoc islands through the gateways’ wireless interfaces. The gateways use their wired interfaces also to communicate with static hosts belonging to a wired LAN. The network resulting from the integration of the hybrid ad hoc network with the wired LAN is an *extended LAN*, in which static and mobile hosts transparently communicate using traditional wired technologies or ad hoc networking technologies. This is achieved by properly employing layer-2 mechanisms, such as the ARP mechanism, and providing additional capabilities to the gateways. Our deployed solution allows logically extending the wired LAN to the ad hoc nodes in a way that is transparent for the wired nodes. In this way, the extended LAN appears to the external world, i.e., the external Internet network, as a single IP subnet, such as that hosts located in the external Internet can

communicate with hosts inside in the extended LAN as they do with traditional Ethernet-based networks.

Several other solutions have been proposed to support Internet connectivity between ad hoc networks, which can be grouped into two general approaches. One approach is to implement a Mobile IP Foreign Agent (MIP-FA) in the ad hoc node that acts as Internet access router, and to run Mobile IP in all the ad hoc nodes [23]. A different approach relies on the implementation of a Network Address Translation (NAT) in the Internet access router, which translates the IP addresses of ad hoc nodes to an address on the NAT gateway, which is routable on the external network [24]. Such approaches are based on complex IP-based mechanisms originally defined for the wired Internet, like the IP-in-IP encapsulation and explicit tunneling, which may introduce significant overheads. On the other hand, the approach we propose in this section is a lightweight solution that avoids these overheads to provide global Internet connectivity to the ad hoc nodes, by exploiting only Layer 2 mechanisms.

4.1 Our solution

To better present our solution (in terms of the rules and operations used to execute the basic services of addressing and routing), throughout this section, we will adopt the reference interconnection scenario depicted in Figure 13. In this scenario we have two ad-hoc islands, which can communicate through a wired LAN. In addition, a node in an ad hoc island can communicate with the Internet through the access router connected to the wired LAN. More details on our approach can be found in [21,22].

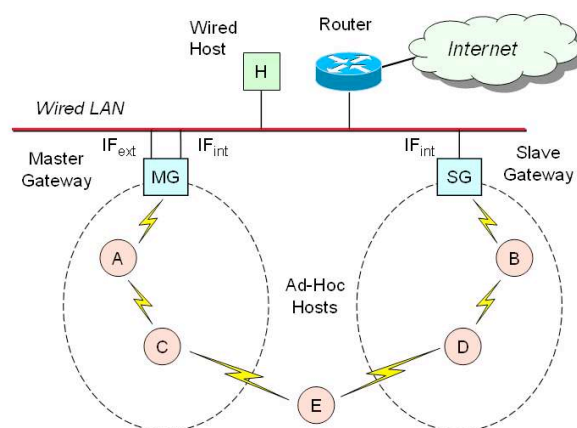


Figure 13: Interconnection scenario

Our solution assumes OLSR as the ad hoc network routing algorithm, although it can be applied to any proactive scheme. Each OLSR-based ad hoc island (i.e., a group of mobile/static nodes communicating through multi-hop wireless links using the OLSR routing protocol) has at least a gateway. We define a gateway as a multi-interface device with both a wireless and one or more wired interfaces. The wireless interface is used to connect the gateway to other ad hoc nodes, while one wired interface is used to connect to other gateways. The OLSR protocol is running on both wireless and wired interfaces to allow the interconnection of different ad hoc islands. Among the gateways, one named the *Master Gateway* (MG) has special capabilities. The MG has two wired interfaces, one of them used to communicate with static hosts belonging to the wired LAN. On the other hand, the other gateways have a single wired interface and are named *Slave Gateways* (SGs). Consequently, the OLSR-based ad hoc network consists of: (i) a set of OLSR-based ad hoc islands, and (ii) a fixed backbone formed by one MG node and multiple SGs connected via wired links. The MG also provides the interconnection between the OLSR-based hybrid ad hoc network and the wired LAN, which provides the connectivity to the external Internet via a default access router.

Our solution assumes that the wired LAN is organized as a single IP subnetwork, i.e., each wired nodes has a static IP address belonging to the same IP subnet, hereafter indicated according to the standard notation IP_s/L , in which IP_s is the IP network address and L is the network mask length (for example $IP_s/L = X.Y.96.0/22$). In this way, the wired hosts are able to exchange packets using their *ARP Table*, which is a list of mappings between the IP address (Layer-3) and the MAC address (Layer-2) of known hosts on the same subnet. The connectivity between the wired LAN and the Internet is provided by a router and standard IP routing protocols.

To ensure a transparent interconnection between the wired LAN and the OLSR-based hybrid ad hoc network, as explained in the following, we assume that ad hoc nodes are configured with static IP addresses belonging to the same subnet of the wired LAN, i.e., IP_s/L . The hybrid nodes' interfaces (i.e., gateways' interfaces) are configured with private IP addresses since they do not generate data traffic but only control traffic (i.e., routing packets). This is also done to facilitate the configuration of the *MG* node, as explained in [22]. It is worth remarking that we devised the *SGs* nodes as simple entities supporting the ad hoc node mobility, and increasing the available bandwidth between the *MG* node and the mobile nodes. For this reason they do not need a globally routable IP address.

The ad hoc routing protocol is implemented by an *OLSRd* daemon, which runs on all the interfaces -wireless and wired -of mobile nodes, *SG* nodes and *MG* node, with the exception of second *MG*-node wired-interface, hereafter denoted as *IF_{ext}*, which provides access to the external network, advertising Internet connectivity as default routes. This is done using special routing messages called *HNA* messages, which inform the ad hoc nodes about the external hosts or networks that can be reached through the *MG*.

Connectivity for Outgoing Traffic. The *OLSRd* daemon builds the routing tables with entries that specify the IP address of the next hop neighbor to contact to send a packet destined to another host or subnetwork. Since the *MG* node advertises $0.0.0.0/0$ as default route, all packets destined for IP addresses external to the IP_s subnet will be routed to the *MG* node and forwarded to the wired LAN and, eventually to the Internet via the default router. On the other hand, when an ad hoc node wants to communicate with another ad hoc node, it will find in the routing table a specific entry indicating the IP address of the next hop neighbor to contact to reach the destination. A special case is when an ad hoc node wants to send a packet addressed to a node on the local wired LAN (e.g., node *H* in Figure 13). In this case, in the routing table there is not a specific entry with the IP address of the next hop neighbor to contact to reach node *H*, but only the default entry that provides $0.0.0.0$ as next-hop IP address for the subnet IP_s/L . This implies that the source node will assume that the destination node is directly connected to the node wireless interface. This will result in a failed *ARP Request* for the IP_H address, sent on the source node's wireless interface. To solve this problem, we have to add to the routing table in each ad hoc node a new entry that forces the traffic destined to the node *H* to be directed towards the *MG*, without using the *ARP* resolution procedures. To achieve this, we observe that the original IP_s/L subnet could be split into two consecutive smaller subnets, $IP_{sl}/(L+1)$ and $IP_{su}/(L+1)$, such as to have $IP_s/L = IP_{sl}/(L+1) \cup IP_{su}/(L+1)$. It is worth pointing out that this operation is always feasible, independently of the particular IP_s/L choice. For example, the subnet $X.Y.96.0/22$ considered at the beginning can be split into the two consecutive subnets $X.Y.96.0/23$ and $X.Y.98.0/23$. Hence, in addition to the default route $0.0.0.0/0$, the *MG* node has to advertise also these two subnets within the *HNA* messages, such that the ad hoc nodes are forced to contact the *MG* to reach hosts belonging to these subnets, and to which they have not more specific routing information. More precisely, each ad hoc node will always have, for any host *H* on the local wired LAN, a routing table entry with a more specific network/mask than the one related to its wireless interface (i.e., IP_s/L). Consequently, the longest-match criterion in the routing table lookup, will determine the right next hop for the IP_H address.

Connectivity for Incoming Traffic. To allow the nodes on the local wired LAN, including the default router, to send packets to the ad hoc nodes, a specific daemon runs on the *IF_{ext}* interface of

the *MG* node, named *Ad Hoc Proxy ARP daemon (AHPAd)*. The *AHPAd* code is reported in Section 6. This daemon periodically checks the Master Gateway's routing table and ARP table, such as to *publish* the *MG* node's MAC address for each IP address having an entry in the routing table with a netmask 255.255.255.255. Indeed, the netmask 255.255.255.255 is the default netmask associated by the OLSR algorithm to each ad hoc node in the network. In this way, the *MG* node will act as a *Proxy ARP* for and only for the ad hoc nodes it is connected to. The entries related to the *IFext* interface of the *MG* node and the *SG* nodes' interfaces are excluded. When an host on the wired LAN wants to send a packet to an ad hoc node, since it considers the destination belonging to the same subnet, initially checks its ARP table for an IP-MAC mapping and, if it is not present, it sends an *ARP Request*. The *MG* node answers with an *ARP Reply* providing its MAC address and the packets can be correctly sent to the destination through the *MG* node.

To conclude some observations are necessary:

- on the *MG* node two wired interfaces are needed because the Proxy ARP does not allow to answer to *ARP Requests* for IP addresses that are reachable through the same interface on which the *ARP Request* was received. Nevertheless, they can be replaced by a single wired interface and emulated by a bridging function and two virtual interfaces.
- The proposed architecture is working only with proactive ad hoc routing protocols. In fact, the Proxy ARP on the *MG* node needs to know all the ad hoc nodes' IP addresses, which has to publish on its ARP Table.

5. REFERENCES

- [1] Charles E. Perkins, Elizabeth M. Belding-Royer and Samir Das, Ad Hoc On Demand Distance Vector (AODV) Routing, IETF RFC.
- [2] P. Jacquet et al, "Optimized Link State Routing Protocol for Ad Hoc Networks", Hipercom Project, INRIA Rocquencourt, BP 105,78153 Le Chesnay Cedex, France.
- [3] Familiar homepage, <http://www.handhelds.org>, March 2003.
- [4] Linux homepage, <http://www.linux.org/>
- [5] GUN's Not UNIX homepage, <http://www.gnu.org/>, February 2003
- [6] Uppsala University AODV homepage, <http://user.it.uu.se/~henrikl/aodv/>, March 2003
- [7] A. Tonnessen, OLSR: Optimized link state routing protocol. Institute for Informatics at the University of Oslo (Norway). Available: <http://www.olsr.org>.
- [8] FreePastry, <http://freepastry.rice.edu>
- [9] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.
- [10] D8 "MobileMAN First Phase", <http://cnd.iit.cnr.it/mobileMAN/pub-deliv.html>
- [11] F. Delmastro, "From Pastry to CrossROAD: Cross-layer Ring Overlay for AD hoc networks", In Proceedings of Mobile Peer-to-Peer 2005 Workshop, in conjunction with PerCom 2005 Conference, Kauai Island, Hawaii, Mar. 2005
- [12] D13 "MobileMAN functionalities - final set", <http://cnd.iit.cnr.it/mobileMAN/pub-deliv.html>
- [13] F. Delmastro and A. Passarella, "An Experimental study of P2P Group-Communication Applications in Real-World MANETs", in Proceeding of IEEE ICPS Workshop on Multi-hop Ad hoc Networks: from theory to reality (REALMAN) in conjunction with IEEE ICPS 2005, Santorini (Greece), July 2005.
- [14] M. Conti, J. Crowcroft, F. Delmastro, A. Passarella, "Cross-Layer Support for Group-Communication Applications in MANETs", in Proceeding of IEEE ICPS Workshop on Multi-hop Ad hoc Networks: from theory to reality (REALMAN) in conjunction with IEEE ICPS 2005, Santorini (Greece), July 2005.
- [15] P. Cremonese, V. Vanni "UDDI4m: UDDI in Mobile Ad Hoc Network", WONS 2005 – short paper.
- [16] GSM codec library, <http://kbs.cs.tu-berlin.de/~jutta/toast.html>
- [17] JRTPLib version 2.9, http://research.edm.luc.ac.be/jori/jrtplib/jrtplib_old.html
- [18] F. Dabek, B. Zhao, P. Druschel, J. Kubiawicz and I. Stoica, "Towards a common API for Structured Peer-to-Peer Overlays", Proc. of the 2nd International Workshop on Peer-to-peer Systems (IPTPS'03), Berkeley, CA, Feb. 2003.

- [19] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "SCRIBE: A large-scale and decentralised application-level multicast infrastructure", IEEE Journal on Selected Areas in Communication (JSAC), Vol. 20, No, 8, October 2002.
- [20] M. Dischinger, "A flexible and scalable peer-to-peer multicast application using Bamboo", Report of the University of Cambridge Computer Laboratory, 2004, available at <http://www.cl.cam.ac.uk/Research/SRG/netos/futuregrid/dischingerreport.pdf>.
- [21] E. Ancillotti, R. Bruno, M. Conti, E. Gregori, A. Pinizzotto, "A Layer 2-based Framework for Implementing Practical Mesh Networks", IIT Technical Report, 2005.
- [22] R. Bruno, M. Conti, E. Gregori, A. Pinizzotto, E. Ancillotti, "Experimenting a Layer 2-based Approach to Internet Connectivity for Ad Hoc Networks", in Proc. of IEEE REALMAN 2005 workshop, Santorini (Greece), 14 July, 2005.
- [23] M. Benzaid, P. Minet, K. Al Agha, C. Adjih, and G. Allard "Integration of Mobile-IP and OLSR for a Universal Mobility", Wireless Networks, 10(4):377-388, July 2004.
- [24] P. Engelstad, A. Tonnesen, A. Hafslund, and G. Egeland, "Internet Connectivity for Multi-Homed Proactive Ad Hoc Networks", in Proc. of ICC'2004, volume 7, pages 4050-4056, Paris, France, June 20-24 2004.

6. APPENDIX: AD HOC PROXY ARP DAEMON SOFTWARE

This section contains the Perl code to implement the gateway (GW) interconnecting ad hoc islands between themselves and with the Internet. The code is divided in two parts: the running code, "ahpad", and a configuration file, ahpad.conf.

A.1 ahpad code

```
#!/usr/bin/perl -w

#####
#
# ahpad - Ad Hoc Proxy Arp Deamon
#
# This daemon periodically reads the routing table and the arp table;
# then adds or deletes "pub" arp entries (used for proxy arp)
# according to the following rule:
#
# for each entry in the routing table with a mask 255.255.255.255
# a "pub" arp entry must be present.
# Some entry can be excluded if specified in the config file.
#
# Usage:
# ./ahpad -f ahpad.conf                # if debug level 1 or 2
# ./ahpad -f ahpad.conf >> ahpad.log & # if debug level 1 or 2
# ./ahpad -f ahpad.conf &              # if debug level 0
#
#
# Known limits:
# this code is just a "prototype"; in fact it uses the output of the
# system
# command "netstat -rn", the syntax of the arp command and the output of
# the
# kernel variable "/proc/net/arp".
# So if these output formats change, for example because of a linux
# release
# upgrade, this program could fail.
#
#
# Created by
# Antonio Pinizzotto <antonio DOT pinizzotto AT iit DOT cnr DOT it>
#
#####
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; either version
# 2 of the License, or (at your option) any later version.
#####

use Getopt::Std;

#####
# config parameter names array
#####

@param_name = (
    "RefreshTime",
```

```
"Interface",
"Exclude",
"DebugLevel",
);

$param_num = @param_name;

$RefreshTime = "";
$Interface = "";
$DebugLevel = 0;

#####
# Other parameters
#####

$ahpadName = "ahpad";

#####
# Time Stamp subroutine
#####

sub time_stamp {
    my
    ($sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst, $dec, $month, $date);

    ($sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst)=localtime(time);
    $dec = int($mday/10); $mday = $dec . ($mday - $dec*10);
    $dec = int($hour/10); $hour = $dec . ($hour - $dec*10);
    $dec = int($min/10); $min = $dec . ($min - $dec*10);
    $dec = int($sec/10); $sec = $dec . ($sec - $dec*10);
    $month = (Gen, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec) [$mon];
    $year += 1900;
    $date = $hour . ":" . $min . ":" . $sec . " " . $mday . " " . $month
    . " " . $year;

    return ("$date");
}

#####

#####
# MAIN
#####

#####
# Get Options
#####

getopts('f:');

if ($opt_f) {
    $CfgFile = $opt_f;
} else {
    print "\n$ahpadName: syntax error: NO configuration file
specified!";
    print "\nCommand Syntax:";
    print "\n<path>\/$ahpadName -f <config_file_name> \&\n\n";
}
```

```

    exit;
}

#####
# Read config file and set parameter values
#####

open(CONFIG, $CfgFile) or die "Can't open $CfgFile: $!\n";
while ($line = <CONFIG>) {
    chop($line);
    $_ = $line;
    while(s/^(\\t| )//) {}          # delete initial tabs or blanks
    $line = $_;
    if (!(($line eq "") || ($line =~ /^#/))) {
        for ($i = 0; $i < $param_num; $i++) {
            if ($line =~ /^$param_name[$i]:/) {
                $var_name = "\\$" . $param_name[$i];
                $pos = index($line, ":");
                $var_value = substr($line, $pos + 1, 10000);
                $_ = $var_value;
                while(s/^(\\t| )//) {}          # delete initial tabs or
blanks
                while(s/(\\t| )$//) {}          # delete final tabs or
blanks
                while (s/"/\\010/) {}          # substitute '"' with '\\"'
                while (s/\\010/\\\\"/) {}      # using "\\010" char
                $var_value = $_;

                if ($var_name eq "\\$Exclude") {
                    $Exclude{$var_value} = "1";
                } else {
                    $command = $var_name . " = " . "\"" . $var_value . "\"";
                    eval($command);
                }
            }
        }
    }
}
close(CONFIG);

#####
# Check for mandatory parameters
#####

if ($RefreshTime eq "") {
    print "\n\n$ahpadName: syntax error: NO RefreshTime value specified
in $CfgFile!\n\n";
    exit;
}

if ($Interface eq "") {
    print "\n\n$ahpadName: syntax error: NO Interface value specified in
$CfgFile!\n\n";
    exit;
}

#####
# If debug level ....
#####

if ($DebugLevel >= 1) {

```

```

print "\n\n+++++";
print "\nAdHoc ProxyArp Deamon ($ahpadName) STARTED";
print "\n+++++\n";
}
if ($DebugLevel >= 2) {
print "-----";
print "\nReading Configuration File \"$CfgFile\"\n";
print "\nRefreshTime = \"$RefreshTime\"";
print "\nInterface = \"$Interface\"";
print "\nDebugLevel = \"$DebugLevel\"";
print "\nExcluded addresses:";
$noone = 1;
foreach $IPAddr (keys %Exclude) {
print "\n $IPAddr";
$noone = 0;
}
if ($noone == 1) {
print " NO ONE";
}
print "\n-----\n";
}
if ($DebugLevel >= 1) {
print "+++++";
print "\nPubArpTable Update Loop STARTED";
print "\nRefresh period: $RefreshTime sec";
print "\n+++++\n";
}

#####
# Loop for "pub arp" update (CORE)
#####

while (1) {

undef @raw_routetable;
undef @raw_arptable;
undef %rt_ip;
undef %at_ip;

# read routing and arp table in raw mode

push (@raw_routetable, `netstat -rn`);
chomp (@raw_routetable);
push (@raw_arptable, `cat /proc/net/arp`);
chomp (@raw_arptable);

# read IP addresses from routing table with a netmask 255.255.255.255
# and load them into keys of an hash array (%rt_ip)

for ($i = 0; $i < @raw_routetable; $i++) {
$_ = $raw_routetable[$i];
while(s/(\t| )/ /) {} # reduce every space to a single
blank while(s/(^ | $)//) {} # delete initial and/or final
blanks $rtline = $_;
if ($rtline =~ /255\.255\.255\.255/) {
undef @tmpfield;
@tmpfield = split (" ", $rtline);
if (defined $tmpfield[2]) {

```



```

        if ($tmpfield[2] eq "255.255.255.255") {
            $rt_ip{$tmpfield[0]} = "";
        }
    }
}

# read IP addresses from arp table with the PUB flag (used for proxy)
# and load them into keys of an hash array (%at_ip)

for ($i = 0; $i < @raw_arptable; $i++) {
    $_ = $raw_arptable[$i];
    while(s/(\t| )/ /) {}          # reduce every space to a single
blank
    while(s/(^ | $)//) {}        # delete initial and/or final
blanks
    $atline = $_;
    if ($atline =~ /0xc/) {
        undef @tmpfield;
        @tmpfield = split (" ", $atline);
        if (defined $tmpfield[2]) {
            if ($tmpfield[2] =~ /0xc/ ) {
                $at_ip{$tmpfield[0]} = "";
            }
        }
    }
}

# update arp table according to the information in
# %rt_ip, %at_ip, %Exclude

$DebugInfo = "";

# add new ip addresses, if necessary

foreach $IPAddr (keys %rt_ip) {
    if ( (!(defined($at_ip{$IPAddr}))) and
(! (defined($Exclude{$IPAddr}))) ) {
        system("arp -s $IPAddr -i $Interface -D $Interface pub");
        $DebugInfo .= "\nADDED    $IPAddr";
    }
}

# delete old ip addresses, if necessary

foreach $IPAddr (keys %at_ip) {
    if ( (!(defined($rt_ip{$IPAddr}))) or (defined($Exclude{$IPAddr}))
) {
        system("arp -d $IPAddr -i $Interface");
        $DebugInfo .= "\ndeleted  $IPAddr";
    }
}

#####
# If debug level ....
#####

if ($DebugLevel >= 2) {
    print "-----";
    print "\n" . &time_stamp() . "\n";
}

```

```

    # print read information
    for ($i = 0; $i < @raw_routetable; $i++) {
        print "\n" . $raw_routetable[$i];
    }
    print "\n";
    for ($i = 0; $i < @raw_arptable; $i++) {
        print "\n" . $raw_arptable[$i];
    }
    print "\n";

    print "\nIP addresses found in routing table (host type)";
    foreach $IPaddr (keys %rt_ip) {
        print "\n$IPaddr";
    }
    print "\n";

    print "\nIP addresses found in arp table (pub type)";
    foreach $IPaddr (keys %at_ip) {
        print "\n$IPaddr";
    }
    print "\n";

    # print changes
    if ($DebugInfo ne "") {
        print "\n*****";
        print $DebugInfo;
        print "\n*****\n";
    }
    print "\n";
}

if ($DebugLevel == 1) {
    if ($DebugInfo ne "") {
        print "-----";
        print "\n" . &time_stamp();
        print $DebugInfo;
        print "\n";
    }
}

#####
# Wait sleeping ....
#####

sleep $RefreshTime;

}

#####
# Eof
#####

```

A.2 ahpad.conf

```
#####
# Configuration file for ahpad (Ad Hoc Proxy Arp Deamon)
#####

# Refresh Time (in seconds)
# The AdHoc ProxyArp Deamon refreshes the arp table every "RefreshTime"
seconds
# No default value; the value MUST be specified

RefreshTime:    1

# Proxy Bind Interface
# The interface on which to bind the proxy arp information,
# that is the interface from which the proxy arp will send arp replies.
# Only ONE interface can be specified
# No default value; the value MUST be specified

Interface:      eth0

# Excluded addresses
# It is possible to exclude some unicast address from being used by the
proxy arp
# More addresses can be specified
# Default: no IP address specified

# Exclude:      192.168.72.22
# Exclude:      192.168.72.33

# DebugLevel
# Possible values:
# 0 - Debug OFF
# 1 - Debug ON
# 2 - Debug ON - VERBOSE
# Debug information are displayed on standard output, that is on the
monitor.
# Obviously, you can redirect it to a file ( >> file).
# Default value: 0

# DebugLevel:   1
```