Helsinki University of Technology

Facultad de Informática - U.P.M.

Design and Implementation of the OLSR Protocol in an Ad Hoc Framework

Helsinki University of Technology **Juan Gutiérrez Plaza**

.

October 2003

To my parents who have always helped me,
to José Costa and Raimo Kantola
who gave me an opportunity of working in this project
and to all my friends who cheered me up continuing.
To every of them, **thanks for doing this possible!**.

A mis padres que me han ayudado siempre,
a José Costa y Raimo Kantola
por darme la oportunidad de trabajar en este proyecto
y a todos mis amigos por animarme siempre a continuar.
A todos ellos, **¡gracias por hacer esto posible!**.

Vanhemmille jotka ovat aina auttaneet minua,
Jose Costalle ja Raimo Kantolalle
jotka antoivat minulle mahdollisuuden työskennellä
tässä projektissa ja kaikille kavereilleni,
jotka kannustivat minua jatkamaan.
Kaikille heille siis suuri kiitos,
koska heidän takiaan tämä kaikki oli mahdollista!.

Acknowlegments.

To everybody who helped me with this Master's Thesis.

Victorino Sanz Prat, José Costa Requena, Ricardo Tormo Mas, Raimo Kantola, Jarrod Creado, David Hernando Vieites, Guillermo de la Calle Velasco, Andrés Moreno García, José Luis López Contell, Oscar Novo Díaz, Céline Papin, Irene Sancho Sánchez, Eduardo García Barrachina, Petra Rubalcaba Redondo, Petteri Saarinen, Lidet Aby, Daniel López, Jose María Vallet, Renjish Kaleelazhicathu, and very specially to Raquel Igelmo Alameda.

Preface

This Master's Thesis has been written at the *Networking Laboratory* of the *Helsinki University of Technology* during a exchange year of the author as Socrates/Erasmus student in this university.

The project is about *Ad Hoc Networking* and shows the maturity and knowledge of the author dealing with a real project and a real problem. Both characteristics were obtained and improved during author's whole university life. But general knowledge needed for developing this project was basically learnt during networking courses and programming courses at author's home university (*Facultad de Informática*, Universidad Politéctica de Madrid -**UPM**-). Specific knowledge of the topic was learnt during developing of the project at the Helsinki University of Technology (**HUT**).

This master's thesis will be delivered to the home university of the author where it will be accepted as a final studies project (rated with 6 UPM credits, that is equivalent to around 4 ECTS credits). This final project is compulsory for obtaining a Spanish Master in Engeneering degree.

A Spanish translation of the Abstract (it is called *Resumen* in Spanish) has also been included for author's home university.

October 2003.
Espoo, Finland.

Juan Gutiérrez Plaza

Abstract

Author:	*Juan Gutiérrez Plaza*
Title of the Thesis:	*Design and Implementation of the OLSR Protocol in an Ad Hoc Framework*
Date: *October 2003*	Number of pages: *78*
Faculty:	*Networking Laboratory* *Helsinki University of Technology (HUT)*
Home Faculty:	*Facultad de Informática* *Universidad Politécnica de Madrid (UPM)*
Supervisor:	*Professor Raimo Kantola*
Instructor:	*MSc. José M. Costa Requena*

Ad Hoc networks are based on a full autonomy of nodes in the network (every node may be a router, gateway, client, server, etc.) and on their capacity for creating networks "per se". Nodes in this kind of networks do not have to be specially configurated and they can find other nodes by themselves.

The work depeloped in this thesis contributes to the implementation of an "Ad Hoc Framework".

The "Ad Hoc Framework" consists of a complete architecture for analysing Ad Hoc networks. It contains several independent routing modules and a common Ad Hoc module.

The common Ad Hoc module consists of a component that allows the execution of multiple routing protocols in the same node. The "Common Registry" and the "Common Cache" are modules of the common Ad Hoc module implemented in this thesis.

The independent routing modules are plug-in components of the "Ad Hoc framework". Following are the modules implemented as part of this thesis that contribute to the "Ad Hoc Framework":

- Independent routing modules:
    - The Optimized Link State Routing protocol (OLSR protocol).
- Common Ad Hoc modules:
    - The Common Registry. It is an information file between protocols.
    - The Common Cache. It is a common routing table which allows to share information.

Implementation choices, improvements, efficiency tests and perfomance of the "Ad Hoc Framework" are reported in the thesis.

This work was partially funded by the Information Society Technologies programme of the European Commision, Future and Emerging Technologies under the IST-2001-3813 MOBILEMAN project.

Keywords: *Ad Hoc, OLSR, framework, proactive protocol*

Resumen

Las redes Ad Hoc se basan en la total autonomía de los nodos (todos y cada uno de ellos pueden actuar como encaminadores, clientes y servidores de cualquier tipo) y en la capacidad de crear una red "per se", es decir, sin tener que configurar algún nodo de manera especial y sin indicarles que otros nodos forman parte de la red (ellos mismos se encangan de buscarse los unos a los otros).

El protocolo llamado OLSR (Optimized Link State Routing Protocol) se ha implementado, probado y en algunos casos mejorado dentro de un marco de trabajo con otros dos protocolos Ad Hoc como son AODV (Ad hoc On Demand Distance Vector) y ZRP (Zone Routing Protocol).

Este marco de trabajo se basa en que estos tres protocolos trabajen conjuntamente en un mismo nodo mejorando así la eficiencia de cualquier protocolo trabajando solo. La mejora propuesta se basa en el diseño de una arquitectura del sistema donde se define un módulo de protocolos independientes y un módulo común del sistema. Este módulo común del sistema incluye un registro común, un módulo de comunicaciones, una lógica de control y en una tabla de rutas común a todos los protocolos (llamada de ahora en adelante *caché común*).

En el *registro común* se guardan las variables de configuración de cada uno de los protocolos así como el rendimiento alcanzado en un momento dado durante su funcionamiento. De esta manera podemos saber cómo maximizar este rendimiento según los valores que toman las variables de configuración de los protocolos en un momento determinado.

El *módulo de comunicaciones*, también llamado CCRS (Common Cache Registry Server), se encarga de manejar los mensajes procedentes de los protocolos que indican si hay un nuevo protocolo corriendo, si se ha encontrado una ruta nueva o bien se ha perdido una existente, etc.

La *lógica de control* es la encargada de elegir apropiadamente los valores de los parámetros influyentes en cada protocolo para optimizar el rendimiento.

Este módulo envia mensajes de control a los protocolos a través del CCRS.

Por último, la *caché común* sólo posee las rutas descubiertas por los protocolos y además información útil para ellos como los servicios prestados por los nodos, qué protocolo ha descubierto la ruta, etc.

En esta tesis se expone y explica el protocolo OLSR, la caché común, el registro de información y el módulo de comunicaciones además de las decisiones de implementación, mejoras realizadas, eficiencia de cada uno de ellos y del conjunto dentro del marco de trabajo así como las pruebas realizadas a cada uno de ellos.

En la parte de implementación se ha desarrollado el protocolo OLSR (basado en el borrador de Internet -"internet draft"- versión 0.7), la caché común, el módulo de comunicaciones y el registro común.

Table of Contents

List of abbreviations and acronyms

AODV	Ad hoc On Demand Distance Vector
ACK	Acknowledge
b	bit
B	Byte
CCRS	Common Cache Register Server
CPU	Central Processing Unit
ECTS	European Credits Transfer System
GNU	GNU is not Unix
GPL	General Public License
HNA	Host and Network Association
HUT	Helsinki University of Technology
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IANA	Internet Assigned Number Authority
iNET	The Internet
LBR	Load Balance Ratio
MANET	Mobile Ad hoc NETworks
MID	Multiple Interface Declaration
OLSR	Optimazed Link State Routing protocol
QoS	Quality Of Service
RAM	Ramdon Access Memory
ROM	Read Only Memory
TC	Topology Control
TTL	Time To Live
UDP	User Datagram Protocol
UPM	Universidad Politécnica de Madrid (Madrid University of Technology)
USB	Universal Serial Bus
ZRP	Zone Routing Protocol

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents information about ad hoc networks and technology. Also the "state of the art" of ad hoc networks is presented in this chapter, the motivation to experiment with new architectures and the objectives of this Master's Thesis.

1.1 Ad hoc networks

Ad Hoc networks are gaining interest day after day as a communications technology. As a result, studies in this field have grown a lot in recent years.

This kind of network without infrastructure started with wireless technology in a practical way. The most important technology has been 802.11x which is cheap and provides high throughput, although other technologies such as Bluetooth can be used.

In essence, every node in the network behaves as a router and all nodes cooperate in carrying traffic [9].

These networks can change their topology very quickly. Adaptability and speed are the two most important characteristics, althouth achieving both is very difficult.

1.2 State of the art

Nowadays, Ad Hoc mobile networking is being studied deeply because it has very important applications in a lot of fields like alternative maritime communications, conferences and congresses and in Military applications (e.g. networking of aircraft, helicopters, tanks and even infanterymen with weareable computers), etc.

Advantages are very clear in these networks, for example:

- Networks without geographical constraints of the fixed networks.
- Flexible topology for a variety of applications.
- There are no wire connections.

Routing is one of the most important problems at this moment.

An efficient and fast routing protocol is very difficult to design because the topology may change quickly. If topology changes a little bit or the number of nodes is not so high, efficient routing can be implementated. But if the number of nodes is high or topology changes a lot, routing will be hard and slow because there will be a lot of messages between nodes for managing this information.

Thus, routing is a hard and common problem that many people try to solve. A lot of protocols and improvements have been proposed by the research community in order to improve the solutions but routing continues to be a difficult issue.

1.3 Motivation

Our proposal is a solution based on the implementation of a prototype of an ad hoc framework [17] to study proactive and reactive proctocols and to explore new algoriths for ad hoc networks.

The Ad Hoc Framework architecture consists of a Common Ad Hoc Module which includes a common cache, common information registry and multiple independent routing modules. The Ad Hoc Framework architecture is based on the concept of *multiprotocol* nodes.

The concept of multiprotocol nodes provides the possibility of creating Ad Hoc networks where some nodes will run different routing protocols but all collaborate towards the lifetime of the Ad Hoc network.

There are multiple alternatives for selecting the apropiate ad hoc routing protocol, reactive, proactive or hybrid:

- Proactive. This kind of protocols are based on getting all routes beforehand.

- Reactive. Routes are searched when they are needed (AODV [16] has been already implemented for this framework).
- Hybrid. These protocols use characteristics of proactive and reactive protocols.

1.4 Objectives of this Master's Thesis

The main objective of this thesis is to implement the OLSR protocol as one of the *independent routing protocols* as a new part of the Ad Hoc Framework.

This thesis also contributes to the framework with the design and implementation of a *common cache*, a protocol parameter storage (*registry*) and a *communication protocol* between the common cache and the independent routing protocols which will be described in chapter 2.

A very important goal achieved in this Thesis is to have two protocols (AODV and OLSR) running on the same node. This feature allows two different networks running different protocols to see each other through the intermediate node (bridge node) which is running both protocols at the same time.

Chapter 2

Theoretical analysis

Prior to every software development, it is compulsory to make a theoretical analysis of the problem, of the solutions and of the ways for arriving to a good solution. This chapter deals with the design of some components of the "Ad Hoc Framework" such as on the independent routing module implementing the OLSR protocol, the Common Cache, the Common Registry and Control Logic.

2.1 Ad Hoc Framework

The Ad Hoc Framework consists of a complete architecture for Ad Hoc networks that include several independent routing modules and a common ad hoc module.

Figure 2.1 shows the modules of the framework and the relationship with the kernel routing table.

Basically, the framework is formed by two modules that includes several components inside:

- Independent routing module.
    1. **Protocols**. These are routing protocols. At the moment there are three (AODV, OLSR and ZRP) but the idea is based on using any implemented protocol.
- Common ad hoc module.
    1. **Common Cache Register Server** (CCRS). This module is a message center. It is listening for register protocols, messages to cache and between protocols etc.
    2. **Registry** is an information file where protocol parameters are stored.

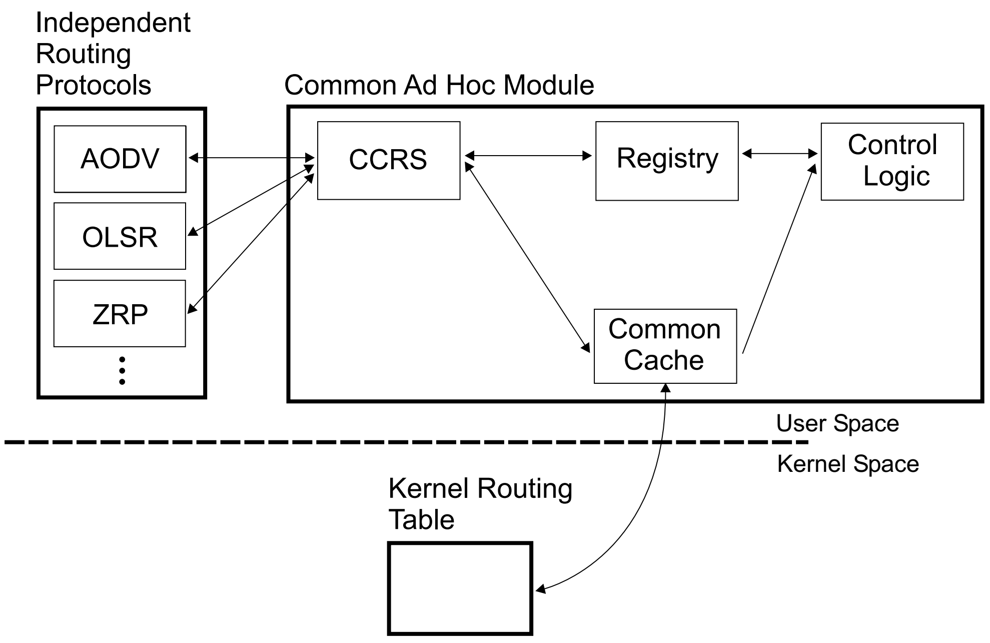

Figure 2.1: Modules of the Ad hoc Framework

3. **Common Cache**. Protocols share routes in this routing table. In this way a node running one protocol can reach other nodes running a different protocol.
4. **Control Logic**. This module reads information from the registry and decides optimal values for protocols parameters, optimal routes, etc.

- External module.

  1. **Kernel Routing Table**. Aplications use the kernel routing table in order to send and forward packets. This table should be coherent with the common cache routing table, this means that at least the kernel routing table must contain all destinations that the common cache has.

The main idea of the framework is to collect information from each protocol in order to get information about the whole network (topology, density, etc.) and to use this information to improve the performance. The performance is improved when the framework changes the protocol parameters and protocols are using optimal values.

This framework also requires that a given node running a protocol within the ad hoc network can send and receive packets of other nodes running an other different protocol.

2.2 The OLSR protocol

2.2.1 Description of the protocol

OLSR is a proactive routing protocol [12], it means that the protocol periodically sends control messages to mantain the knowledge of the network topology. OLSR protocol is a "link state" protocol, this means a node broadcasts over the network the list of its neighbors. In this case all the nodes know the neighborhood of all the nodes. Therefore, the nodes have all the routes and thus the sorthest path to all the destinations.

This protocol sends several kinds of control messages in order to mantain the knowledge of the network and to know the neighborhood.

The OLSR protocol is an optimization of a pure link state protocol. This optimization is based on two premises. First, it reduces the size of control packets: instead of all links, it declares only a subset of links with its neighbors who are named as *Multipoint Relay Selectors*. Secondly, it minimizes flooding of this control traffic by using only selected nodes, *Multipoint Relays* to diffuse its messages in the network.

Two types of control messages are handled by the OLSR protocol:

1. Neighbour sensing.
2. Topology discovery.

Prior to explaining the control messages, two conceps must be introduced:

- **Multipoint Relays** (*MPR*). The idea behind the MPR concept is to minimize the flooding of broadcast packets. Each node in the network will select a set of neighbouring nodes, which will retransmit its packets through the network (see Figure 2.2). The set of selected neighbouring nodes is called MPRs of that node [12].
- **Multipoint Relay Selectors** (*MPR Selectors*). All the neighbouring nodes that select a given node as an MPR belong to a set called MPR Selectors set. A given node only broadcast packets sent by nodes belonging to its MPR Selector set.

Neighbour sensing messages are called *HELLO* messages. These kind of control messages are used to build the neighbourhood surrounding a node

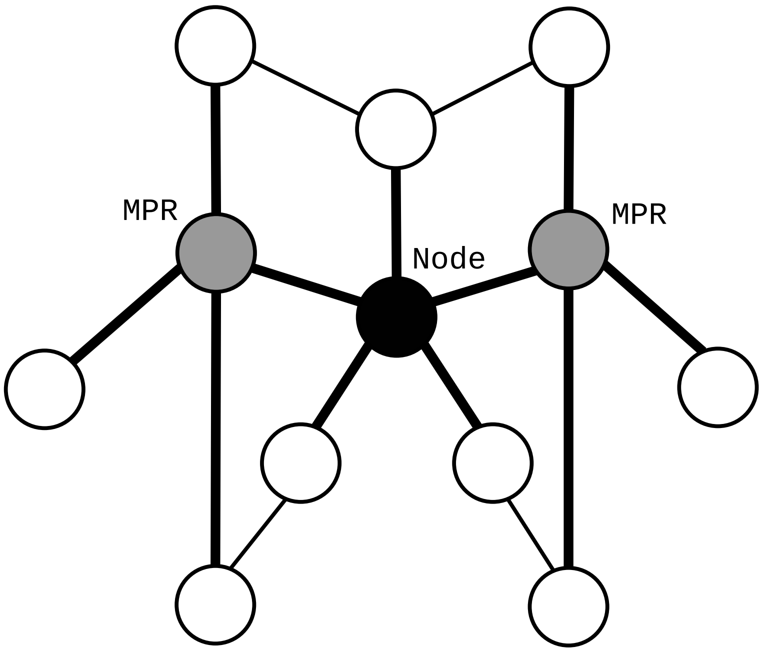

Figure 2.2: Multipoint relays

(only neighbours which can communicate directly with the node belong to the neighbourhood) and also to compute the MPRs of a node [13]. HELLO messages are sent in broadcast packets to nodes which are one hop away from the original node.

Topology discovery is used to know the topology of the whole network and it handles several kinds of messages:

- **Topology Control** (TC). This is the most important type of message for topology discovery. A TC message is sent periodically by each node in the network to declare its *MPR Selector* set.
- **Multiple Inteface Association** (MID). These messages inform about the nodes which have more than one MANET interface.
- **Associated Networks and Hosts** (HNA). Nodes with more than one non-MANET interface broadcast this kind of message.
- **Fast Re-Routing** (FRR). These messages send information about routing.

These messages are broadcast through the whole network but they use MPRs in order to minimize the flooding of the network.

2.3 Common cache

2.3.1 Introduction

The benefit of using a common cache is to share routing information between protocols. This way, a node can collect information about other nodes running a different protocol. If this was not so, nodes running a single protocol are blinded to the information collected by nodes running other protocols.

We will also benefit from having a common cache by collecting other information from it such as metrics, cost, services provided by the node, etc. This extra information will help a node to decide whether to choose a given node as a router or to choose a different one.

There are two well defined different data structures for implementing an efficient cache. One of them is a **Search Tree** (and any kind of its subtype, Binary Tree, Ternary Tree, B-Tree, etc.) and the other one is a **Hash Table**.

2.3.2 Brief description of data structures

Search Trees

Basically, a search tree is a data structure where searching takes place using by a specific field, called a *key*. Every subtree of a node has values less than any other subtree of the node to its right. The values in a node are conceptually between subtrees and are greater than any values in subtrees to its left and less than any values in subtrees to its right.

The key field is presented on each and every node and is unique. One or more values of the key are possible in a node depending whether the tree is binary, ternary or quaternary (see Figure 2.3).

Nodes are reached by branches from their parent node.

There are balanced trees (as binary trees) and unbalanced trees (as red-black trees, AVL-trees or B-trees). Their characteristics are:

- **Unbalanced**. These trees do not dynamically change their structure to improve their access time. Instead, they have less computational load and need less time for insertions and deletions. The time needed for seaching an entry for the worst case is $\Theta(n)$ (where n is the depth of the tree).
- **Balanced**. In this case, trees change dynamically to improve the searching access time. When a deletion or insertion occurs, the tree

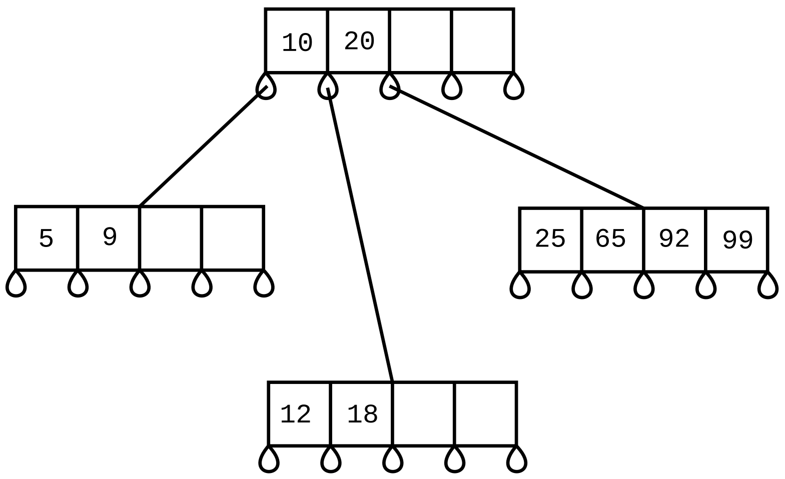

Figure 2.3: Example of a quaternary tree

changes and all nodes, if possible, are set to have the same number of branches with the minimum depth. The search takes $\Theta(log_2 n)$ time for the worst case.

Hash Table

This kind of structure is based on a dictionary in which keys are mapped to array positions by a hash function. Having more than one key map to the same position is called a collision. There are many ways to resolve collisions, but they may be divided into open addressing, in which all elements are kept within the table, and chaining, in which additional data structures are used.

Insertions and deletions are simple and they do not need a lot of time and computational load. Searches are very fast; the searching time depends on the hash function and collision resolution (see Figure 2.4), but may be constant ($\Theta(1)$) although the worst case is $\Theta(n)$.

2.3.3 Pros & Cons

B-Trees

Choosing a balanced tree:

- **Pros**.
    1. Maximun complexity $\Theta(log_2 n)$.
    2. No collisions.
    3. Fast.

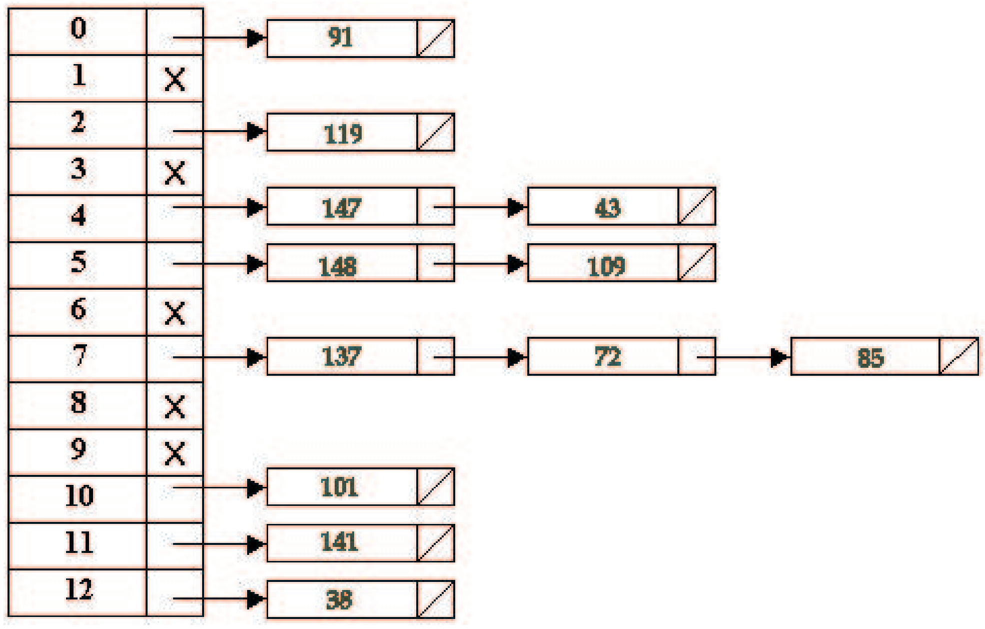

Figure 2.4: Example of hash table which resolves collisions with link lists

4. Minimizes memory access.

- **Cons**.

  1. Insertion and deletion are complex and slow.
  2. Complex structure.

Hash Table

- **Pros**.

  1. In most cases complexity is constant $\Theta(1)$.
  2. Really good behavior for dictionary applications (insert, delete and search).
  3. Very fast if it uses a good hash function.
  4. Insertation and deletion are simple.
  5. Simple structure.

- **Cons**.

  1. Very slow in the worst case.
  2. Slow and large for a lot of data.

2.3.4 Hash Table

According to the analysis of both alternatives, the hash table was chosen to implement the *common cache*. The hash table works well when implementing the routing cache (not so large amount of entries and the data is clearly differenciated by a key).

If the hash function is well defined, the searching time is constant and very fast and insertions and deletions are simple.

We need a simple hash table API for our purposes. The API is as follows.

- Create table.
- Insert data.
- Delete data.
- Search data.
- Destroy table.

And optionally, we could add:

- Update data.
- Update field.
- Get field of an entry.

Following, table 2.1 shows the fields proposed for the common cache, the size of the fields in bits and some comments to explain each field.

Fields	Size (b)	Comments
Type	8	Each bit set stands for an active protocol
Destination IP type	1	IPv4 or IPv6
Destination address	32 or 128	IPvx of destination
Next Hop IP type	1	IPv4 or IPv6
Next Hop address	32 or 128	IPvx of destination
Time stamp	32	Time of creation or last update
Cost	8	Cost to reach next hop
Metrics	24	8 bits for battery, 8 for signal and 8 for QoS
Services	16	Remote services which are running (DNS, DHCP, etc.)
Location	48	Geographic location (x, y, z)
Fqdm	undefined	Names of the node

Table 2.1: Fields proposed for the common cache

Hash table efficiency

Assuming an Ad Hoc network composed of 1000 nodes, it is clear that all nodes will have a different IP address.

Given the described scenario with a hash table of 521 entries we can store 521 nodes without using a chained list. With a two nodes chaining list, we can store 1042 nodes and with a 3 nodes chaining list, 1563 nodes. In a not so good case, we can store 1000 nodes with chaining lists of two and three elements. In the best case, hash table can store 1000 nodes with chaining list of one and two elements but hashing function has to be perfect (it has to spread the key perfectly uniformnly) and node IP addreses that are used for generating the key must be good.

The size of the hash table is $521 \cdot 4 = 2084B$ (entry pointers), $1.000 \cdot 4 = 4000B$ (maximum data pointer) and $1000 \cdot 4 = 4000B$ (data key comparation), therefore the table (without data) is $\simeq 10KB$ size in the worst case.

Types of hash tables

There are several kinds of hashing. The two most important are:

- **Open hashing**. If there is a collision, a link list is built. Therefore, the hash table may have a list per entry.
- **Close hashing**. There are no lists. If there is a collision, a second hashing function is used (may be a hashing function or search the next free entry, for example).

In the first case, any number of data can be stored, although access time increases. In the second case, it is not possible because we have to know beforehand how much data is allowed.

For implementing a routing table we can only choose *open hashing* because we cannot know the number of nodes, it means that we cannot know number of route entries in the routing table.

Conclusion

Implementing a common routing table with a link lists hash table is the best choice. The main reasons are that searching complexity may be constant, insertions and deletions are simple and the amount of data is not very large. Also it is good that the memory size is not big. Double hashing, for example, is not a good choice since we do not know ahead of time how many nodes could store the table.

Therefore, we conclude that this kind of structure for implementing the common cache is the most suitable approach.

2.4 Common registry

2.4.1 Introduction

Common registry is a component of the *common ad hoc module* within the Ad Hoc framework that will save information on several protocols in the same node. This registry saves specific information from each protocol and common information as *last running time*, *status of the protocol*, *most efficient configurarion information of protocol running last time*, etc.

This information is used by *Control Logic* in order to improve on the routing efficiency. The control logic calculates performace of each protocol and chooses the best configuration for each of them according to the actual network status.

2.4.2 Brief description of data structures

Firstly, we have to differenciate the common information from the specific information.

1. **Common information**. Consist of information stored in the registry, which is common for all prococols running in the node.

    - *Status.* Status of the protocol. It can be "running alone", "running with olsr", "running with aodv" or "running with zrp".
    - *Time last stop.* This field is only written if protocol is stopped. It shows when protocol was stopped.
    - *IP type.* It can be IPv4 or IPv6. If protocol is not running this field shows the last IP type used.

2. **Protocol specific information**.

    (a) **AODV information**.

    - *Active route timeout.*
    - *Allowed hello loss.*
    - *Hello interval.*
    - *Net diameter.*
    - *Node traversal time.*
    - *Rerr rate limit.*
    - *Rreq retires.*
    - *Rreq rate limit.*
    - *TTL increment.*
    - *TTL threshold.*

    (b) **OLSR information**.

- *Number of hops.*
- *HELLO interval.*
- *TC interval.*
- *MID interval.*
- *HNA interval.*
- *Dup interval.*
- *Neighb hold time.*
- *Neighb 2 hop hold time.*
- *TC hold time.*
- *MID hold time.*
- *HNA hold time.*
- *TC redundancy.*
- *MPR coverage.*
- *Willing.*
- *Purge interval.*

(c) **ZRP information**.

- *Zone Radious.*
- *IARP protocol.*
- *IERP protocol.*
- *BRP status.*
- *NDP status.*
- *Target LBR.*
- *Current LBR.*
- *RZC update.*
- *Broadcast TP.*
- *Table refresh.*
- *Node density.*
- *Node speed.*

2.5 Control logic

This Ad Hoc Framework subsystem calculates the best configuration of the node in real time. Control logic reads common information and private information of each protocol running (this information can be read from common registry) and decides which is the best value for a private variable in order to maximize the performance.

It can initialize a protocol if performance can be improved. For example, when AODV is running and almost all the packets are sent to destination nodes which are one hop distance away from the original node, if the time of entry has expired then this routing entry is deleted and the next access to this node requires AODV to search for the path again. This is a very bad case for AODV, however for OLSR it is a good case because nodes which are one distance hop away are always in the routing table. Therefore, in this scenario, control logic would initialize OLSR protocol in order to improve the one hop distance away access.

Control logic is not only based in physical measurements. Statistics are also very important for taking the best way for improving.

This module is not implemented yet and it is out of the scope in this thesis.

2.6 Communication between subsystems

Every subsystem within the Ad Hoc Framework has to communicate with each other in order to work efficiently (see Figure 2.5). The ad hoc routing protocols must communicate with the common cache and the common registry.

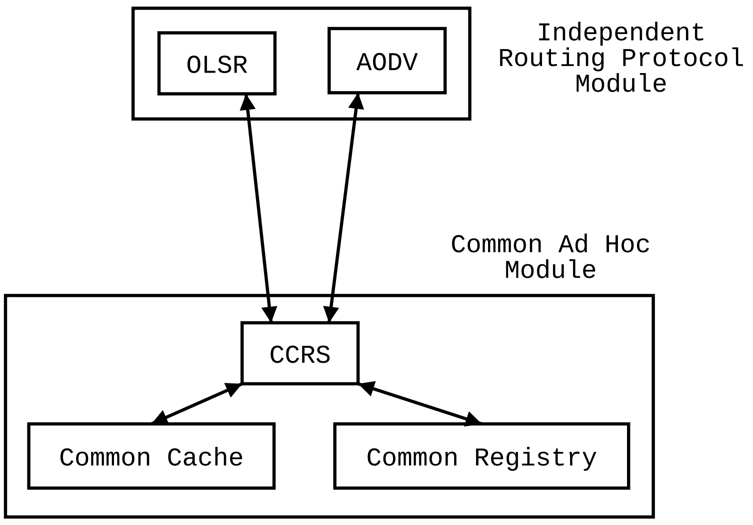

Figure 2.5: Communication between subsystems of the framework

The module called CCRS provides this communication.

The first prococol running must initialize the CCRS. Then, the process of the CCRS listens (at a pre-defined port) for each protocol and routing access.

When a protocol wants to be registered, it sends a port number where it will listen to. Then, CCRS knows that this port will be used only for that protocol.

Figure 2.6 shows steps for registering a protocol:

1. A protocol sends a registry message to the CCRS by a common port (where CCRS is always listening). This message contains the number of the port where the protocol will send and receive messages.

2. CCRS searches for an old configuration of the protocol in the common registry.

3. CCRS sends an ACK to the protocol to the given port by the protocol with the old configuration if it exists.

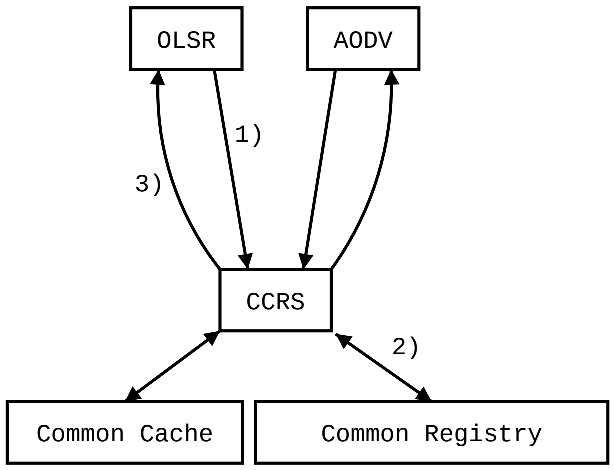

Figure 2.6: Communication protocol

Chapter 3

Development

This chapters shows the software development in this project. OLSR implementation, common cache, common registry and control logic are explained. Implementation choices, improvements and eficiency are described as well.

3.1 OLSR protocol

Our OLSR implementation follows v.0.7 OLSR iNET draft [10] and it has been started from the beginning.

It uses UDP protocol for comunicating and the port 698 assigned by IANA.

The OLSR protocol implementation has been written from scratch althouth other alternatives were studied as [11]. However, this implementation is based on v.0.3 of OLSR iNET draft. Our implementation have followed the moludarity, although, because performance in iPAQs, the implementation is at low level.

The size of our implementation is around 3000 lines of code and the memory footprint is 3864 B.

This protocol is formed by several modules. Figure 3.1 shows module diagrams of the system. It also shows the interdependency between modules and where external framework modules are needed.

- **init**. This module initializes data structures, gets information of the node, configures sockets options, etc., and "executes" the protocol.

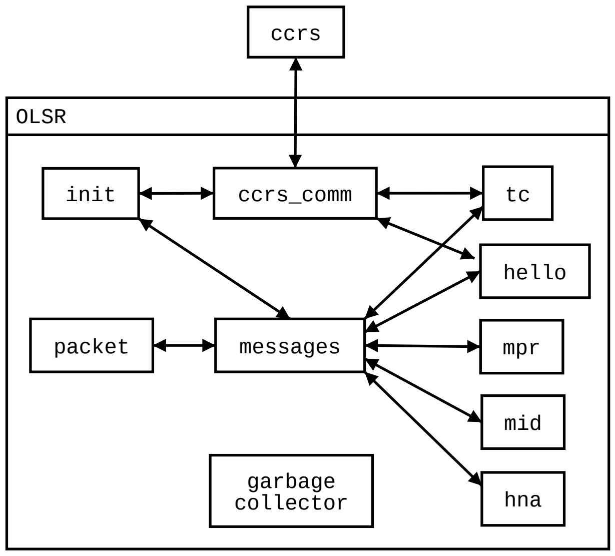

Figure 3.1: Module diagram of the OLSR protocol

- **packet**. Packet module is a low level module which works building packets with message information and sending them. This module reads packets and saves the information in OLSR packet structures.
- **garbage collector**. It reads all lists used by the protocol and deletes old entries.
- **messages**. This module speaks directly with the packet module and other specific message format modules. It contains the high level messages processing.
- **ccrs_comm**. This is an API to communicate with the ccrs program.
- **tc**. It is the module which generates and processes topology control messages.
- **hello**. This module generates and processes HELLO messages.
- **mid**. This module generates and processes multiple interface declaration messages.
- **hna**. It is the module which generates and processes host and network association messages.
- **mpr**. This module works with MPR and MPR Selector.

Lists are a very important part in this protocol. Lists are used in a lot of modules to store control messages information.

All lists in the system have been implemented as follows:

```
typedef struct {
  struct list *first;
  struct list *last;
} list_t;

typedef struct {
  struct in_addr main_addr;
  struct in_addr if_addr;
  struct in_addr if2_addr;
  struct in_addr main2_addr;
  unsigned short valid_time;
} listEntry_t;

struct list {
  listEntry_t *entry;
  struct list *next;
  struct list *prev;
};
```

This is a list with two pointers; one of them points to the first element of the list and the other points to the last element. Also, a next pointer and a previous pointer have been used. These two additional pointers (pointer to last element and previous pointer) help to manage lists easier and adding and deleting are faster than usual. We can see a graphical diagram of the list in Figure 3.2.

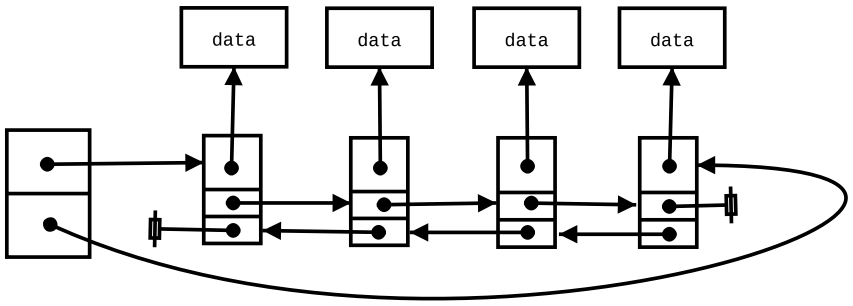

Figure 3.2: The list used in OLSR implementation

3.2 Common cache

In this section we explain the most suitable way for implementing a *Common Cache* or *Common Routing Table* which is being used by several Ad Hoc protocols simultaneoustly.

Hash table is the selected (see Section 2.3) approach for implementing the common cache.

The hash table consists of a collection of key-value pairs. The key of each pair is something uniquely associated with the corresponding value. For our common cache this key is the IP address because they are uniquely associated with a node.

When the hash function receives a key, it has to transform the key into a valid index for the table (in our default case a number between 0 and 520). This transformation is obtained as follows:

$$index = key \bmod 521$$

In case of IPv6, we cannot directly access the table by an IP address because our hash function must have the same kind of input. Figure 3.3 shows how we have resolved this problem by doing an XOR for each four bytes in a IPv6 address.

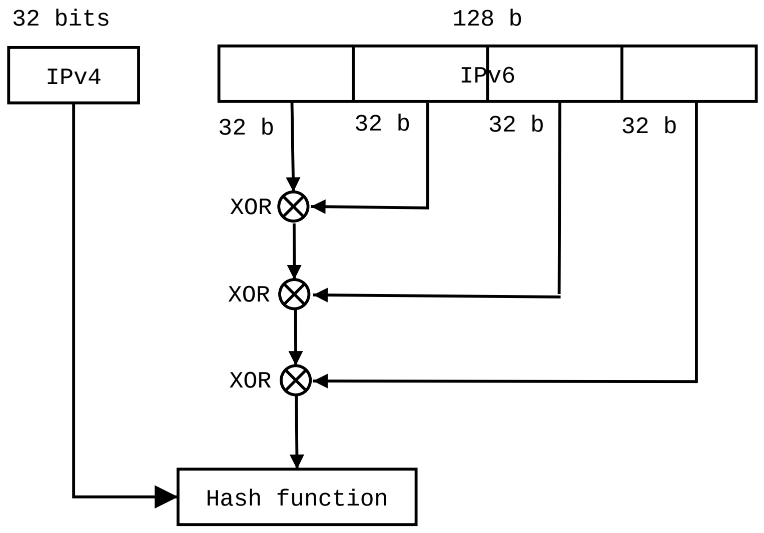

Figure 3.3: Input in the hash function

3.2.1 Fields and structures

The specific data structures used for implementing the hash table are shown in Appendix B.

3.2.2 API

There are two differents APIs, one for local access and another for external access (i.e. external access from "independent routing protocols").

Local access.

- **Create Hash Table**.
  - Parameters.
    1. Number of entries (integer).
  - Return value. Hash table (pointer).
- **Delete Hash Table**.
  - Parameters.
    1. Hash table (pointer)
  - Return value. Void.
- **Hash**. Get the key from an entry.
  - Parameters.
    1. Hash table (pointer)
    2. IP (unsigned integer).
  - Return value. Key (unsigned int).
- **Add Hash**. Add a new entry to the Hash Table.
  - Parameters.
    1. Hash table (pointer)
    2. IP (unsigned int)
    3. Data (pointer to struct data)
  - Return value. New entry (pointer)
- **Find Hash**. Find an entry in the Hash Table.
  - Parameters.
    1. Hash table (pointer)
    2. IP (unsigned int)

3. Protocol (enumerate). Values may be: AODV, OLSR, ZRP or ANY_PROT

- Return value. Searched route (pointer) or NULL.

- **Delete Hash**. Delete an entry in the Hash Table.
  - Parameters.
    1. Hash table (pointer)
    2. IP (unsigned int)
    3. Protocol (enumerate)
    4. Is route in kernel? (interger). Out parameter.
  - Return value. Error or OK (integer).

External access.

- **New Route**.
  - Parameters.
    1. Address.
    2. Gateway (next hop).
    3. Mask.
    4. Cost.
    5. Services.
    6. Metrics.
    7. Coordenates.
    8. Names.
    9. Input interface.
  - Return value. ACK.
- **Delete Route**.
  - Parameters.
    1. Address.
    2. Gateway (next hop).
  - Return value. ACK.
- **Update Route**.
  - Parameters.
    1. Address.
  - Return value. ACK.

3.2.3 Update policy

The implementation of the data structure, which is used in the common cache entries, includes a *time_stamp* field, which enables the possibility of applying certain cache policy (e.g. most entries used or most recently entries used). This means that the kernel routing table will only store a reduced number of entries. The number of entries in the common cache will always be greater or equal than the number of kernel routing entries, which must be lower because of performance reasons.

However, this implementation is quite hard and it is out of the scope of this Master's Thesis.

The policy implemented is based only on "cost". Every reachable node has a single route entry in the kernel routing table but probably more than one in the common cache. However, the cache entries copied in the kernel route table will always be the ones with the minimal cost from the ones available in the common cache.

Common cache stores all routes whether or not the cost is minimumal.

Therefore, when a route is discovered, it is written into the cache and if it has the minimum cost it is also entered into the kernel.

When a route is lost, this route is deleted from the kernel table (if it exists here) and from the cache. Then an alternative minimun cost route is searched in the cache and written into the kernel routing table.

For implementing this simple algorithm we have added a new field in the cache. This field is called *kernel* and shows whether the route is only in the cache (with a "0" value) or if it is in the kernel as well (with a "1" value).

3.3 Common registry

The common registry has an API with only two functions:

```
int proc_request (u_int8_t type, prot_t prot, void *mens);
int read_conf (prot_t prot, reg_t *conf /* out */);
```

In the first function, "type" is the type of the operation which can be:

1. RE_REQ_MSG. Register protocol.
2. UNRE_REQ_MSG. Unregister protocol.

The second paramenter called "prot",which identifies the type of protocol that wants to register or unregister (in our case it only can be AODV, OLSR or ZRP).

The last paremeter is a pointer which points to a region of memory where the parameters of the protocol are stored.

The second function reads the actual or the last (it depends if protocol is running or not) valid configuration of a protocol and returns this value in the reference paremeter "conf".

Configuration information is stored in an ascii file called *prots.config*. Writing the configuration in a file facilitates debbuging although it is not the optimal way. In this file the following information is written:

- Time of creation.
- Table with possible protocol status.
- Protocols running or protocols which were running in the past (with last stop time).

The following lines show an example of this file.

```
Common Cache Registry.
Created: Wed Jun 11 19:43:22 2003

Protocol status:
R_alone = 0
R_ZRP = 1
R_OLSR = 2
R_AODV = 3

ZRP Radius: 2 IARP: olsr IERP: aodv, BRP_status: 1 NDP_status: 1 tlbr: 0.600000
clbr: 0.450000 rzcupd: 10 broadcast: 10 table_refresh: 15 node_d: 8 node_speed: 150
IPv: 4 Status: 1 Time_last_stop: Thu Jun 19 13:01:48 2003
OLSR Num_Hops: 2 Hello_Refresh: 10 Valid_Time: 5 IPv: 4 Status: 0
```

3.4 Control logic

The implementation of "Control Logic" is out of the scope of this thesis.

3.5 Communication between subsystems

The size of the implementation of communication between subsystems is around 2000 lines of code. It includes the CCRS, the common cache and the registry modules. The memory footprint for this communication (CCRS process) is 1680 B.

Communication between subsystems have been implemented with UNIX named sockets. Sockets are local to the machine and they do not have external communication.

The common registry process has been implemented to listen to ten protocols at the same time (this number can be changed at this line within the implementation).

```
listen (aodv_fd, 10);
```

The ccrs server must listen to one socket for the protocol registration (the port name "/tmp/adhoc" is well known by all protocols) and to another socket for each concrete protocol.

When a protocol sends a message to the register itself, this message also contains the new socket ("/var/tmp/aodv", "/var/tmp/olsr"...) for future communications between the protocol and the server. Finally, the server sends back an ACK to the protocol and also starts to listen to this new socket.

The flow diagram is shown in Figure 3.4.

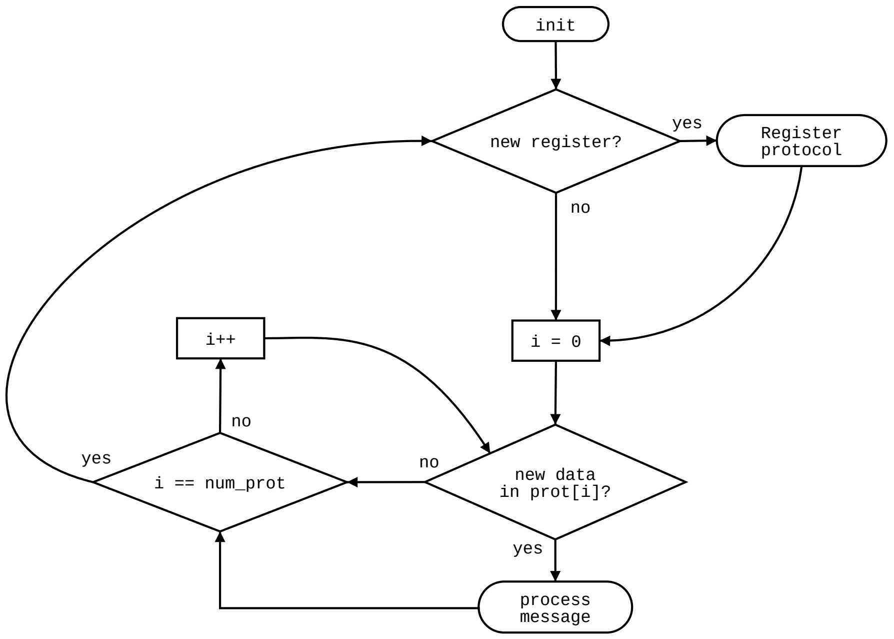

Figure 3.4: CCRS server flow diagram in the registration operation

In Figure 3.5, we can see the transaction diagram of the registration operation.

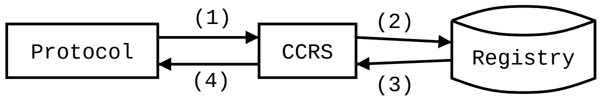

Figure 3.5: The transaction diagram of a protocol registration

1. Upon protocol initialization, the protocol registers to the CCRS by sending a `REG_MSG` (containing the structure of the socket that the protocol will listen to) through the named socket "/tmp/adhoc".
2. CCRS uses one of the registry API functions to examine the registry file for a valid configuration of the protocol.
3. Either valid protocol configuration string is returned from the file or it is indicated that no configuration is available.
4. CCRS replies with an ACK to the new socket where the protocol is already listening. The configuration (from the registry) is piggybacked on the ACK, if available.

In this new socket, the protocol send petitions to the CCRS as follows:

- New route discovered.
- Search for a route.
- Delete a route.
- Get the configuration of another protocol.
- Put the configuration of another protocol.

CCRS can send different messages to protocols:

- New route discovered by another protocol.
- Delete route discovered by another protocol.
- Change configuration.

Transaction diagrams of these operations are the following:

- New route discovered.

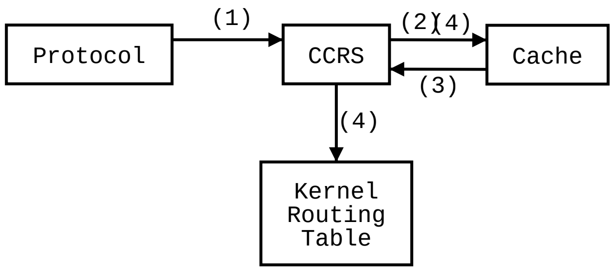

Figure 3.6: Transaction diagram of a new route discovered

(1) The protocol uses its own named socket ("/var/tmp/[protocol]") to send a `NEW_ROUTE_MSG` to CCRS.

(2) Then CCRS searches if the route already exists in the cache, in the cache and the kernel or in none of them.

(3) The reply contains the existence of the route: in the cache, in the cache and kernel or in none of them.[1]

(4) If the route does not exist, then the route is written in the cache and the kernel.

- Search a route. Similar to previous case.
- Delete a route. Similar to "new route" case.
- Get configuration.

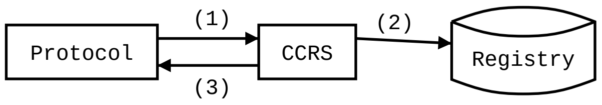

Figure 3.7: Transaction diagram of "get configuration"

(0) ZRP or other similar protocol (hybrid) checks if its sub-protocol (reactive or proactive) is running. If not, it starts the relevant protocol.

(1) ZRP sends the message `GET_CONFIG` through its named socket ("/var/tmp/[hybrid_protocol]") to the CCRS, containing the relevant protocol name, indicating which configuration it wants to retrieve.

(2) Configuration is retrieved from the registry file.

[1]At the same time, CCRS sends a message with the new route to the rest of the protocols telling that another protocol has discovered a route in order to the prococols can update their internal data structures.

(3) ACK back to hybrid protocol with the configuration information.

- Set configuration.

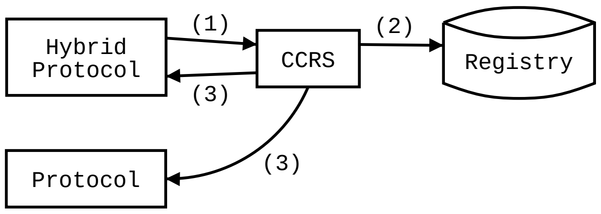

Figure 3.8: Transaction diagram of a “set configuration”

(0) Having previously executed a `GET_CONFIG` procedure, a hybrid protocol has manipulated the configuration and now it wants to write it back to the registry.

(1) A hybrid protocol sends the message `SET_CONFIG` through its named socket to the CCRS containing the relevant protocol name and indicating which configuration it wants to set.

(2) Configuration is written to the registry file.

(3) ACK back to the hybrid protocol and a message type called `CHANGE_CONFIG` (containing the piggybacked new configuration from the hybrid protocol) is issued to the relevant protocol through the named socket “/var/tmp/[protocol]”.

The CCRS server can listen to one socket at a same time giving priority to the register socket.

Code of CCRS server is shown in appendix A.

Chapter 4

Scalability tests, analysis and integration

This chapter deals with the evaluation of the Ad Hoc Framework implementation and its integration. Tests are planned and documented to evaluate the performance of the implemented modules. The aim of the tests is to analyze how the protocol behaves.

4.1 Configuring *iPAQ 3950*

This framework has been tested with real wireless nodes. We have used five *iPAQ 3950* running a GNU/Linux operating system (Familiar distribution [1]) and a laptop running a RedHat distribution with a wireless card. Installation of the operating system, configuration of the iPAQ, compilation and instructions of the installation of protocols, CCRS, etc., have been included as appendixes (see Appendix F for more information).

4.2 Tests

All tests have been done sending *ping* messages between nodes. The number of nodes was six, five iPAQs and one laptop. All the nodes have only one MANET interface and do not have any other type of interface. Links and connections between nodes change in each test.

We have done the same test with OLSR running alone and OLSR running with AODV in the same node.

The configuration of OLSR is described in Appendix D.

The configuration of AODV is the default configuration of this protocol commented in [16].

The test use cases are the following:

- Test 1. Fully-meshed nodes running only OLSR.
- Test 2. Nodes aligned within node range coverage running only OLSR.
- Test 3. Nodes grouped and connected through a single node.
  - Nodes with OLSR.
  - Intermediate node with AODV+OLSR and border nodes with AODV or OLSR.

4.2.1 Test 1

This is the best case for OLSR. All nodes are at a distance of one hop and all links are direct links.

Configuration

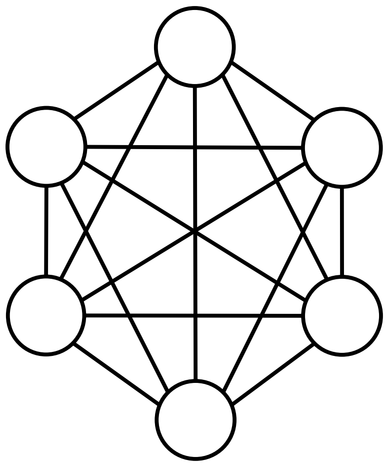

Figure 4.1: Configuration of test 1, fully meshed nodes

OLSR running alone

With this configuration, OLSR shows an excellent behaviour. Routes are quickly discovered (all of them are discovered by all the nodes within the first 7 seconds because it is the time needed to receive the first HELLO message from another node) and they are holding without any problem.

No node has lost packets with this configuration. In the following, some statistics of "ping" are shown for one node. Test was done when all nodes were pinging to all nodes.

```
64 bytes from 10.0.0.6: icmp_seq=45 ttl=255 time=2.954 msec
64 bytes from 10.0.0.6: icmp_seq=46 ttl=255 time=3.225 msec
64 bytes from 10.0.0.6: icmp_seq=47 ttl=255 time=3.039 msec
64 bytes from 10.0.0.6: icmp_seq=48 ttl=255 time=2.462 msec
64 bytes from 10.0.0.6: icmp_seq=49 ttl=255 time=2.922 msec
64 bytes from 10.0.0.6: icmp_seq=50 ttl=255 time=3.587 msec

--- 10.0.0.6 ping statistics ---
51 packets transmitted, 51 packets received, 0% packet loss
round-trip min/avg/max = 2.114/3.117/9.605 ms
```

Kernel route table was modified as follows:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.5        *               255.255.255.255 UH    0      0        0 eth1
10.0.0.4        *               255.255.255.255 UH    0      0        0 eth1
10.0.0.6        *               255.255.255.255 UH    0      0        0 eth1
10.0.0.1        10.0.0.5        255.255.255.255 UGH   0      0        0 eth1
10.0.0.3        *               255.255.255.255 UH    0      0        0 eth1
10.0.0.2        *               255.255.255.255 UH    0      0        0 eth1
127.0.0.0       *               255.0.0.0       U     0      0        0 lo
```

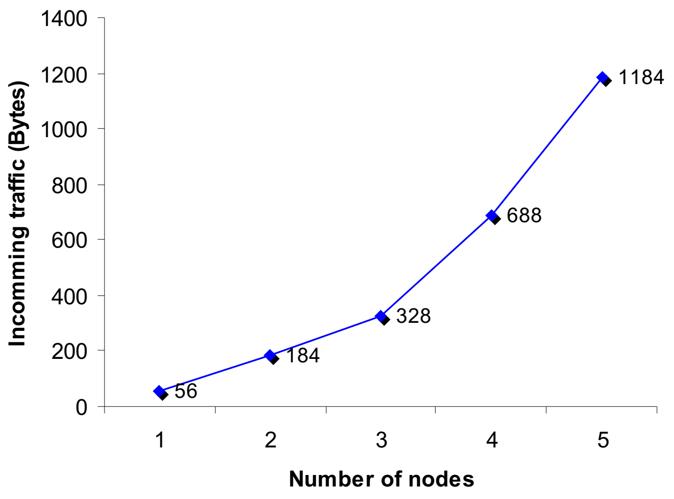

Figure 4.2: Traffic for test 1

However, the common cache has more route entries than the kernel routing table as we can see in table 4.1 (log file of the cache is added in Appendix C).

Destination	Gateway	Cost	Kernel
10.0.0.5	*	1	1
10.0.0.6	*	1	1
10.0.0.6	10.0.0.5	2	0
10.0.0.1	10.0.0.5	2	1
10.0.0.5	10.0.0.6	2	0
10.0.0.1	10.0.0.6	2	0
10.0.0.2	*	1	1
10.0.0.2	10.0.0.6	2	0
10.0.0.2	10.0.0.5	2	0
10.0.0.1	10.0.0.2	2	0
10.0.0.5	10.0.0.2	2	0
10.0.0.6	10.0.0.2	2	0
10.0.0.4	*	1	1
10.0.0.5	10.0.0.4	2	0
10.0.0.6	10.0.0.4	2	0
10.0.0.2	10.0.0.4	2	0
10.0.0.4	10.0.0.5	2	0
10.0.0.1	10.0.0.4	2	0
10.0.0.4	10.0.0.6	2	0
10.0.0.4	10.0.0.2	2	0
10.0.0.3	*	1	1
10.0.0.3	10.0.0.6	2	0
10.0.0.5	10.0.0.3	2	0
10.0.0.6	10.0.0.3	2	0
10.0.0.3	10.0.0.2	2	0
10.0.0.3	10.0.0.4	2	0
10.0.0.3	10.0.0.5	2	0
10.0.0.3	10.0.0.5	2	0
10.0.0.2	10.0.0.3	2	0

Table 4.1: Route entries in the common cache -Test 1-

When a route is lost, another alternative route is searched. For example, if the node looses the 10.0.0.2 route, the route is deleted from the cache and the kernel and another route as 10.0.0.2 $\rightarrow$ 10.0.0.3 will be written in the cache.

The down fall of this configuration with this protocol is the high traffic of control messages. Traffic increases expotetially with each new node.

We can see the traffic as a function of the number of nodes in Figure 4.2 This traffic is the routing protocol incoming traffic per a given node.

OLSR uses MPR nodes and these nodes filter some TC messages. This protocol has an "already processed messages" list in order to filter more messages. As a result, traffic is less than in an original (no optimized) reactive protocol.

4.2.2 Test 2

It was difficult to test this configuration and we had to repeat the test several times.

In most cases links were broken for a moment due to the building structure. There were a lot of interferences (other wireless networks and wireless phones) and signal power and range were often changing.

In Figure 4.3 we can see the location of nodes for the test.

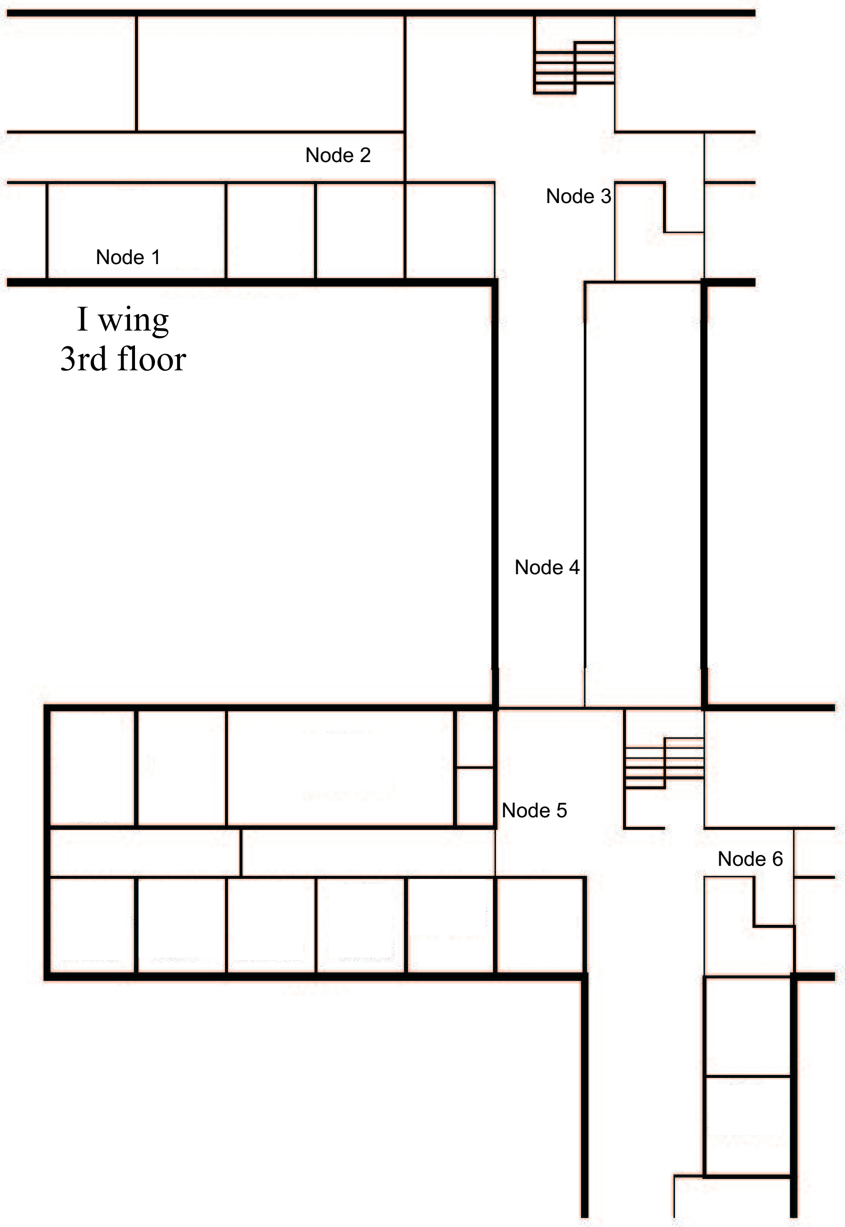

Figure 4.3: Nodes location for test 2

Ping messages have been sent between first and last nodes (10.0.0.1 and 10.0.0.6).

Configuration

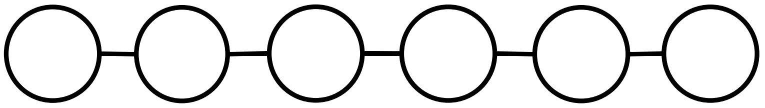

Figure 4.4: Configuration of test 2, nodes aligned within node coverage range

OLSR running alone

Routes in this case are discovered quite fast (between 0 and 15 seconds all the routes are established in all the nodes), but it is notably slower than the previous configuration. The reason for this behaviour is that now TC messages also create some routes in addition to similar routes created with the HELLO messages.

TC messages create routes "out of local range", this means that there are no direct links.

The following are ping test results:

```
64 bytes from 10.0.0.6: icmp_seq=45 ttl=251 time=72.6 msec
64 bytes from 10.0.0.6: icmp_seq=46 ttl=251 time=23.9 msec
64 bytes from 10.0.0.6: icmp_seq=47 ttl=251 time=25.3 msec
64 bytes from 10.0.0.6: icmp_seq=48 ttl=251 time=19.9 msec
64 bytes from 10.0.0.6: icmp_seq=49 ttl=251 time=22.2 msec
64 bytes from 10.0.0.6: icmp_seq=50 ttl=251 time=21.1 msec

--- 10.0.0.6 ping statistics ---
51 packets transmitted, 45 packets received, 12% packet loss
round-trip min/avg/max = 19.4/27.7/72.6 ms
```

Some packets were lost due to broken links in a given instant. Other worse tests showed broken links for more time and for this reason some routes were deleted from the route tables (common cache and kernel). In these cases the protocol needed more time in order to find the route again.

This bad behavior is caused by interferences and by the building structure as discussed before.

For the first node there is only one direct route, the rest of them are indirect and the next hop is always the second node (10.0.0.2). The routing table for first node is as follows:

```
Kernel IP routing table
```

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.2        *               255.255.255.255 UH    0      0        0 eth1
10.0.0.3        10.0.0.2        255.255.255.255 UH    0      0        0 eth1
10.0.0.4        10.0.0.2        255.255.255.255 UH    0      0        0 eth1
10.0.0.1        10.0.0.2        255.255.255.255 UGH   0      0        0 eth1
10.0.0.5        10.0.0.2        255.255.255.255 UH    0      0        0 eth1
10.0.0.6        10.0.0.2        255.255.255.255 UH    0      0        0 eth1
127.0.0.0       *               255.0.0.0       U     0      0        0 lo
```

The common cache routing table has all the possible routes given by TC messages received from other nodes.

Traffic is less than in the previous case. The HELLO messages do not arrive to all the nodes and some TC messages are filtered by MPR. Traffic is a bit heavier in internal nodes (because they receive HELLO messages and TC messages from two nodes) than in the peripheral nodes.

4.2.3 Test 3

OLSR running alone

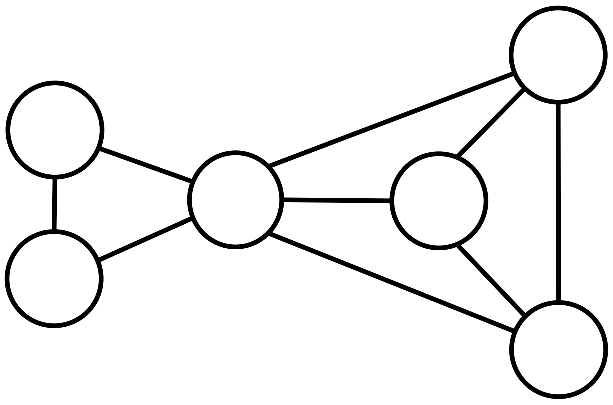

Figure 4.5: Configuration of test 3, two sets of nodes connected through a single intermediate node

This test has been performed with static nodes and dynamic nodes.

- **Static**. The behavior in this case is quite good, time between peripheral nodes is less than 15 ms and routes in all nodes and MPR set are well formed. Less than 2% of all packets were lost. We added the log of TC messages of the central node as Appendix E.

- **Dynamic**

1. The initial configuration is test 3 configuration and the final configuration is test 1 configuration.
   In this case, new routes (all the routes in the new configuration are direct) are discovered quickly, in 3 seconds all the new routes are in the routing table and old routes are deleted.
2. The initial configuration is test 1 configuration and the final configuration is test 3 configuration.
   The behaviour in this case is worse than in the previous one. Routes are discovered slower than in static case because the node waits until the entries of direct nodes in the neighbour set have expired. When these entries are deleted, new entries are written in the route table inmediately because routes were discovered previously thanks to TC messages.
   In some cases we have found some broken links and the stable configuration was reached later.
   The log file of the central node about general behavior has been included in Appendix E.

In this case we have not included the "ping" results because there are no "extreme cases" like in the previous tests.

In Appendix E we can see all logs for this test.

OLSR running with AODV

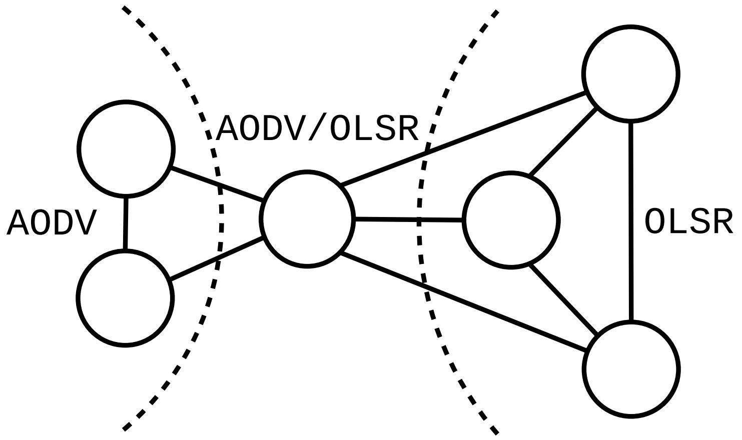

Figure 4.6: Configuration of test 3 running OLSR and AODV

Both protocols run perfectly together in one node. The common cache is correctly updated and registration of protocols is fine.

But this test has had some problems because the nodes of AODV network cannot see the nodes of the OLSR network.

The problem for AODV is that ICMP packets must be captured by the protocol and checked to see if the destination has been written in the common cache by another protocol. This characteristic is out of the scope of this Master's Thesis.

Chapter 5

Conclusions and future work

This chapter deals with the final results of this project, evalutations, achievements, future work, improvements, innovations, etc.

5.1 Conclusions

This OLSR implementation works quite well with static nodes, although its behaviour is worse when nodes are moving. This is the expected behaviour according to simulations. Consequently, the protocol works as well as simulations [18] lend to expect.

We have checked that the protocol does not work fine with several "jumps" between nodes (see test 2). We cannot know if this behaviour is caused by bugs in the code or interferences. There are many factors that affect this kind of communication: physical environments, interferences, etc. To be sure of this behaviour is caused by other factors and not by implementation bugs, we should make some more exhaustive tests of the code ("white box tests") although a lot of tests have been already performed during the implementation and testing phases.

Although we have not tested the framework with several networks running different protocols (see test 3), the common cache, the CCRS and the registry have had an excellent behaviour in a node running two different protocols (OLSR and AODV). Therefore, we can conclude that the cooperation between AODV and OLSR using the common cache, the CCRS, etc., improves the performance of each protocol running alone and provides new features for the network.

5.2 Future work

The next step in this project would be to design and implement the Control Logic module. A good design of this control logic (probably based on probability and statistics) will let the whole system run better and reach an optimal state. The main objective in further study would be to verify if this improvement on the system is feasible. This means that control logic is expected to have a great computational load (because during execution it has to read variables from the common registry and calculate optimal values) and this improvement cannot be provided sustainably by devices with reduced computational power (e.g. iPAQ) as compared with the current desktop PC standars. For compensating this computational load, the control logic should calculate optimal values when performace parameters are out of a defined range. This means that control logic is not always running and it would work only under certain circunstances.

The framework has been designed and implemented to work with other protocols in addition to OLSR, AODV and ZRP. Therefore, another future work would be to test the framework with new protocols.

The information in the control registry is in an ascii file saved on permanent store (hard disk, flash, etc.). The access to the fields of this file is sequential and this is not the optimal way. An improvement to this problem could be made if this file was a memory file (binary file). When the control registry process is initializated, it should take this file and save it in the main memory. Every access or modification to the file would be directly done on memory. If a new protocol is initializated, it will read the information file from the main memory and not from disk. Finally, the common registry process should often save the infomation file on disk and the same should happen if the process is stopped.

Finally, studying the cooperation of AODV and OLSR with and without ZRP in the framework and the performance of these protocols and the common modules (the CCRS, the control logic, etc) would be other tasks for a future work.

Bibliography

[1] http://www.handhelds.org

[2] http://handhelds.org/feeds/BootBlaster3900/BootBlaster3900.exe

[3] http://handhelds.org/feeds/bootldr/pxa/bootldr-2.19.57.bin

[4] http://handhelds.org/ pb/unstable/bootopie-pb6a-h3900.jffs2

[5] http://h20022.www2.hp.com/busprod/overview/0,12512,series=96474
%5Etype=64929%5Ecategory=215383,00.html?lsidebarLayId=106&rsidebarLayId=63

[6] ftp://ftp.handhelds.org/pub/linux/arm/toolchain/monmotha

[7] http://www.tct.hut.fi/˜xlei/ipaq/download/ipkg-make-kernel-packages

[8] http://www.tct.hut.fi/˜xlei/ipaq/download/ipkg-build

[9] "Ad Hoc Networking", Published in "Systems", pp 33-40. *Carlo Kopp*, Febrary, 2002

[10] "Optimized Link State Routing Protocol", Internet Draft v0.7 - IETF MANET Working Group, *Thomas Clausen, Philippe Jacquet et all*, 1st of November of 2002.

[11] http://hipercom.inria.fr/olsr/olsrd.tar.gz

[12] "Optimized Link State Routing Protocol for Ad Hoc Networks", *P. Jacquet et all*, Hipercom Project, INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France.

[13] "Simulation Result of the OLSR Routing Protocol for Wireless Network", *Anis Laouti et all*, INRIA Rocquencourt, 78153 Le Chesnay Cedex, France.

[14] "Unix Network Programming. Networking APIs: Sockets and XTI", *W. Richard Stevens*, Volume 1 - Second Edition, Pretice Hall, 1998.

[15] "Ad Hoc Networking", *Charles E. Perking*, Addison Wesley, 2001.

[16] "Design and implementation of an Ad Hoc routing framework", *Lei Xiao*, Master's Thesis, Helsinki University of Technology.

[17] "Replication of routing tables for mobility management in ad hoc networks", *José Costa Reuena, Nicklas Beijar* and *Raimo Kantola*. Accepted to ACM Wireless Networks (WINET) Journal, 2003.

[18] "Simulation Results of the OLSR Routing Protocol for Wireless Network", *Anis Laouti*, *Paul Mhlethabler* et al, INRIA Rocquencourt, 78153 Le Chesnay Cedes, France.

Appendix A
Communication between subsystems

When a daemon receives a message or several messages from an internal socket, it has to queue messages and serve one at a time. Listening all sockets and serving one are implemented by a "while", "select" and "for" sentences. Code is as follows:

```
...
  while (1) {
    memcpy ((char *) &rfds, (char *) &readers, sizeof(rfds));

    if ((n = select (nfds + 1, &rfds, NULL, NULL, NULL)) < 0) {
      if (errno != EINTR)
        printf("Failed select (main loop)");
      continue;
    }

    if (FD_ISSET (listenfd, &rfds)) {
      if ((clifd = ccrs_socket_accept (listenfd, &type)) < 0)
        printf("server accept error %d\n", clifd);
      i = client_add (clifd, type);
      FD_SET (clifd, &readers);
      if (clifd > nfds)
        nfds = clifd;
      if (i > clients)
        clients = i;
      printf("new connection: type %d, fd %d\n", type, clifd);
      continue;
    }

    for (i = 0; i <= clients; i++) {
      if ((clifd = client[i].fd) < 0)
        continue;
      if (FD_ISSET (clifd, &rfds)) {
...
```

“While” always returns to “select” which is listening at all ports. Then, if one or several ports have information then a “for” looks each protocol ports. Finally with a “FD_ISSET” we can know if this port has ready information.

Appendix B
Fields and structures of common cache

The next structure is typical for implementing a hash table with link list chaining. There are two pointers to the same structure (they are only used if a collision exists), one field for data (it is a pointer to a data structure) and another one for the key.

```
struct htab {
  struct htab *child;  /* Child if collision */
  struct htab *parent; /* Parent if collision */
  unsigned long key;   /* Key of hash function, 32 bits */
  struct data *data;   /* Routing entry */
};
```

The data structure of the cache entry is where information is saved. We have changed IP version field from one bit to one byte because accessing to memory is faster. IP addresses are inside a “union”, it means that memory used is the biggest of both (in this case 128 bits).

```
struct data {
  u_int8_t type;
  u_int8_t vdest;
  union ip_dest {
    struct in_addr  dest_ipv4;
    struct in6_addr dest_ipv6;
  };
  u_int8_t vnhop;
  union ip_nexthop {
    struct in_addr  nhop_ipv4;
    struct in6_addr nhop_ipv6;
  };
  u_int8_t time_stamp;
  u_int8_t cost;
  struct metric *metrics;
  u_int16_t_t services;
  struct coords *coords;
  u_int8_t kernel;
  struct fqdm *fqdm;
};
```

Metric structure is very simple. It is only three fields of one byte.

```
struct metric {
  u_int8_t battery;
  u_int8_t signal;
  u_int8_t qos;
};
```

Coordenates structure is also quite simple. It is only three fields of two bytes that represent x, y and z coordinates.

```
struct metric {
  u_int16_t x;
  u_int16_t y;
  u_int16_t z;
};
```

Names of a node are stored in a link list. We know the end of the list when the “next pointer” points to “NULL”.

```
struct fqdm {
  char *name;
  struct fqdm *next;
};
```

Appendix C

Configuration of OLSR for tests

Parameter	Value
Number of hops	255
Hello interval	7
TC interval	5
MID interval	5
HNA interval	5
Dup interval	30
Neighb hold time	17
Neighb 2 hop hold time	17
TC hold time	13
MID hold time	13
HNA hold time	13
TC redundancy	2
MPR coverage	1
Willing[1]	3
Purge interval	10

Table 1: Default values for OLSR

1

[1] For all nodes this value is the standard, 3, but for laptop we have used 7

Appendix D
Log of the common cache

Example for test 1.

```
new connection: type 1, fd 4
a request received:
this is a registry request
type = 1, protocol = 1
A new node has been discovered
Ip: 10.0.0.5
Nh: 0.0.0.0
A new node has been discovered
Ip: 10.0.0.6
Nh: 0.0.0.0
Add new route to the cache
Ip: 10.0.0.6
Nh: 10.0.0.5
A new node has been discovered
Ip: 10.0.0.1
Nh: 10.0.0.5
Add new route to the cache
Ip: 10.0.0.5
Nh: 10.0.0.6
Add new route to the cache
Ip: 10.0.0.1
Nh: 10.0.0.6
A new node has been discovered
Ip: 10.0.0.2
Nh: 0.0.0.0
Add new route to the cache
Ip: 10.0.0.2
Nh: 10.0.0.6
Add new route to the cache
Ip: 10.0.0.2
Nh: 10.0.0.5
Add new route to the cache
Ip: 10.0.0.1
Nh: 10.0.0.2
```

```
Add new route to the cache
Ip: 10.0.0.5
Nh: 10.0.0.2
Add new route to the cache
Ip: 10.0.0.6
Nh: 10.0.0.2
A new node has been discovered
Ip: 10.0.0.4
Nh: 0.0.0.0
Add new route to the cache
Ip: 10.0.0.5
Nh: 10.0.0.4
Add new route to the cache
Ip: 10.0.0.6
Nh: 10.0.0.4
Add new route to the cache
Ip: 10.0.0.2
Nh: 10.0.0.4
Add new route to the cache
Ip: 10.0.0.4
Nh: 10.0.0.5
Add new route to the cache
Ip: 10.0.0.1
Nh: 10.0.0.4
Add new route to the cache
Ip: 10.0.0.4
Nh: 10.0.0.6
Add new route to the cache
Ip: 10.0.0.4
Nh: 10.0.0.2
A new node has been discovered
Ip: 10.0.0.3
Nh: 0.0.0.0
Add new route to the cache
Ip: 10.0.0.3
Nh: 10.0.0.6
Add new route to the cache
Ip: 10.0.0.5
Nh: 10.0.0.3
Add new route to the cache
Ip: 10.0.0.6
Nh: 10.0.0.3
Add new route to the cache
Ip: 10.0.0.3
Nh: 10.0.0.2
Add new route to the cache
Ip: 10.0.0.3
Nh: 10.0.0.4
Add new route to the cache
```

```
Ip: 10.0.0.3
Nh: 10.0.0.5
Add new route to the cache
Ip: 10.0.0.2
Nh: 10.0.0.3
Add new route to the cache
Ip: 10.0.0.4
Nh: 10.0.0.3
Add new route to the cache
Ip: 10.0.0.1
Nh: 10.0.0.3
this is a unregistry request
type = 4, protocol = 1
closed : fd 4
```

Appendix E
Logs in test 3

OLSR running alone

TC log

Central node is 10.0.0.1.

In incoming TC messages are shown:

- Message sender. `sender: ....`
- Message originator. `orig: ....`
- Time stamp.
- Neighbour of originator. `neigh: ....`
- If message is forwarded by current node. `^--- Forward msg`

```
...

TC MSG: sender: 10.0.0.5, orig: 10.0.0.5 t: Fri Sep 19 11:27:00 2003
neigh: 10.0.0.1
neigh: 10.0.0.6
neigh: 10.0.0.3
^--- Forward msg

TC MSG: sender: 10.0.0.6, orig: 10.0.0.1 t: Fri Sep 19 11:27:02 2003
neigh: 10.0.0.5
^--- Forward msg

TC MSG: sender: 10.0.0.6, orig: 10.0.0.6 t: Fri Sep 19 11:27:03 2003
neigh: 10.0.0.1
neigh: 10.0.0.5
neigh: 10.0.0.3
neigh: 10.0.0.2
^--- Forward msg
```

```
TC MSG: sender: 10.0.0.2, orig: 10.0.0.2 t: Fri Sep 19 11:27:03 2003
neigh: 10.0.0.1
neigh: 10.0.0.6
neigh: 10.0.0.3
neigh: 10.0.0.5
neigh: 10.0.0.4

TC MSG: sender: 10.0.0.3, orig: 10.0.0.3 t: Fri Sep 19 11:27:04 2003
neigh: 10.0.0.1
neigh: 10.0.0.6
neigh: 10.0.0.2
neigh: 10.0.0.5
neigh: 10.0.0.4

TC MSG: sender: 10.0.0.5, orig: 10.0.0.5 t: Fri Sep 19 11:27:05 2003
neigh: 10.0.0.1
neigh: 10.0.0.6
neigh: 10.0.0.3
^--- Forward msg

TC MSG: sender: 10.0.0.5, orig: 10.0.0.1 t: Fri Sep 19 11:27:05 2003
neigh: 10.0.0.6
neigh: 10.0.0.4
^--- Forward msg

TC MSG: sender: 10.0.0.2, orig: 10.0.0.2 t: Fri Sep 19 11:27:06 2003
neigh: 10.0.0.1
neigh: 10.0.0.6
neigh: 10.0.0.3
neigh: 10.0.0.5
neigh: 10.0.0.4

TC MSG: sender: 10.0.0.4, orig: 10.0.0.4 t: Fri Sep 19 11:27:07 2003
neigh: 10.0.0.1
neigh: 10.0.0.3
neigh: 10.0.0.6

TC MSG: sender: 10.0.0.5, orig: 10.0.0.1 t: Fri Sep 19 11:27:08 2003
neigh: 10.0.0.6
neigh: 10.0.0.4
neigh: 10.0.0.2
^--- Forward msg

TC MSG: sender: 10.0.0.6, orig: 10.0.0.6 t: Fri Sep 19 11:27:08 2003
neigh: 10.0.0.1
neigh: 10.0.0.5
neigh: 10.0.0.3
neigh: 10.0.0.2
```

```
^--- Forward msg

TC MSG: sender: 10.0.0.3, orig: 10.0.0.3 t: Fri Sep 19 11:27:09 2003
neigh: 10.0.0.1
neigh: 10.0.0.6
neigh: 10.0.0.2
neigh: 10.0.0.5
neigh: 10.0.0.4

...
```

In outcoming TC messages are shown:

- Time stamp.
- Neighbour of current node. **`neigh: ....`**

```
...
TC MSG Sal
neigh: 10.0.0.6 t: Fri Sep 19 11:27:11 2003
neigh: 10.0.0.5 t: Fri Sep 19 11:27:11 2003
neigh: 10.0.0.4 t: Fri Sep 19 11:27:11 2003
neigh: 10.0.0.2 t: Fri Sep 19 11:27:11 2003

TC MSG Sal
neigh: 10.0.0.6 t: Fri Sep 19 11:27:14 2003
neigh: 10.0.0.5 t: Fri Sep 19 11:27:14 2003
neigh: 10.0.0.4 t: Fri Sep 19 11:27:14 2003
neigh: 10.0.0.2 t: Fri Sep 19 11:27:14 2003
neigh: 10.0.0.3 t: Fri Sep 19 11:27:14 2003

TC MSG Sal
neigh: 10.0.0.6 t: Fri Sep 19 11:27:17 2003
neigh: 10.0.0.5 t: Fri Sep 19 11:27:17 2003
neigh: 10.0.0.4 t: Fri Sep 19 11:27:17 2003
neigh: 10.0.0.2 t: Fri Sep 19 11:27:17 2003
neigh: 10.0.0.3 t: Fri Sep 19 11:27:17 2003

TC MSG Sal
neigh: 10.0.0.6 t: Fri Sep 19 11:27:20 2003
neigh: 10.0.0.5 t: Fri Sep 19 11:27:20 2003
neigh: 10.0.0.4 t: Fri Sep 19 11:27:20 2003
neigh: 10.0.0.2 t: Fri Sep 19 11:27:20 2003
neigh: 10.0.0.3 t: Fri Sep 19 11:27:20 2003
...
```

General log

Information showed by this log is the following:

- If a income message is duplicate. Message originator and sequence number.
- Nodes add to TC set.
- New direct neighbours. This line is always written when a HELLO message is received. For this reason route may alredy exist.
- New neighbours at 2 hops. It's the same case than direct neighbours.
- New mpr nodes. When two hop neighbourhood changes, the mpr set is deleted and the protocol starts searching for the nodes which will formed the new mpr set.
- New mpr selector nodes.
- When a mpr selector is deleted.

```
...
Duplicate message. Orig: 10.0.0.5, Seq: 132
Duplicate message. Orig: 10.0.0.6, Seq: 145
Duplicate message. Orig: 10.0.0.5, Seq: 134
Add TC. Dest: 10.0.0.3, Last: 10.0.0.3, Mssn: 0 :: Time: Fri Sep 19 11:26:58 2003
Duplicate message. Orig: 10.0.0.6, Seq: 146
New neighb on 2 hop. If: 10.0.0.6, Main: 10.0.0.6
                     If2: 10.0.0.3, Main2: 10.0.0.3 :: Time: Fri Sep 19 11:26:59 2003
Add MPR. Main: 10.0.0.6
Add TC. Dest: 10.0.0.3, Last: 10.0.0.3, Mssn: 0 :: Time: Fri Sep 19 11:27:00 2003
Duplicate message. Orig: 10.0.0.5, Seq: 135
New neighb on 2 hop. If: 10.0.0.5, Main: 10.0.0.5
                     If2: 10.0.0.3, Main2: 10.0.0.3 :: Time: Fri Sep 19 11:27:02 2003
Add MPR. Main: 10.0.0.6
New neighb on 1 hop. If: 10.0.0.4, Main: 10.0.0.4,
                     Willing: 3 :: Time: Fri Sep 19 11:27:02 2003
Duplicate message. Orig: 10.0.0.6, Seq: 148
Duplicate message. Orig: 10.0.0.6, Seq: 148
Add TC. Dest: 10.0.0.3, Last: 10.0.0.3, Mssn: 0 :: Time: Fri Sep 19 11:27:03 2003
Add TC. Dest: 10.0.0.5, Last: 10.0.0.5, Mssn: 0 :: Time: Fri Sep 19 11:27:03 2003
Add TC. Dest: 10.0.0.4, Last: 10.0.0.4, Mssn: 0 :: Time: Fri Sep 19 11:27:03 2003
Add TC. Dest: 10.0.0.1, Last: 10.0.0.1, Mssn: 0 :: Time: Fri Sep 19 11:27:04 2003
Add TC. Dest: 10.0.0.6, Last: 10.0.0.6, Mssn: 0 :: Time: Fri Sep 19 11:27:04 2003
Add TC. Dest: 10.0.0.2, Last: 10.0.0.2, Mssn: 0 :: Time: Fri Sep 19 11:27:04 2003
Add TC. Dest: 10.0.0.5, Last: 10.0.0.5, Mssn: 0 :: Time: Fri Sep 19 11:27:04 2003
Add TC. Dest: 10.0.0.4, Last: 10.0.0.4, Mssn: 0 :: Time: Fri Sep 19 11:27:04 2003
Duplicate message. Orig: 10.0.0.5, Seq: 137
Duplicate message. Orig: 10.0.0.5, Seq: 137
Duplicate message. Orig: 10.0.0.5, Seq: 137
Add TC. Dest: 10.0.0.4, Last: 10.0.0.4, Mssn: 0 :: Time: Fri Sep 19 11:27:05 2003
Duplicate message. Orig: 10.0.0.1, Seq: 224
New neighb on 2 hop. If: 10.0.0.6, Main: 10.0.0.6
```

```
                     If2: 10.0.0.2, Main2: 10.0.0.2 :: Time: Fri Sep 19 11:27:06 2003
Add MPR. Main: 10.0.0.6
Duplicate message. Orig: 10.0.0.2, Seq: 26
Add TC. Dest: 10.0.0.1, Last: 10.0.0.1, Mssn: 0 :: Time: Fri Sep 19 11:27:07 2003
Add TC. Dest: 10.0.0.3, Last: 10.0.0.3, Mssn: 0 :: Time: Fri Sep 19 11:27:07 2003
Add TC. Dest: 10.0.0.6, Last: 10.0.0.6, Mssn: 0 :: Time: Fri Sep 19 11:27:07 2003
New neighb on 1 hop. If: 10.0.0.2, Main: 10.0.0.2,
                     Willing: 3 :: Time: Fri Sep 19 11:27:07 2003
New neighb on 2 hop. If: 10.0.0.2, Main: 10.0.0.2
                     If2: 10.0.0.6, Main2: 10.0.0.6 :: Time: Fri Sep 19 11:27:07 2003
Add MPR. Main: 10.0.0.6
Add MPR. Main: 10.0.0.2
New neighb on 2 hop. If: 10.0.0.2, Main: 10.0.0.2
                     If2: 10.0.0.3, Main2: 10.0.0.3 :: Time: Fri Sep 19 11:27:07 2003
Add MPR. Main: 10.0.0.6
Add MPR. Main: 10.0.0.2
New neighb on 2 hop. If: 10.0.0.2, Main: 10.0.0.2
                     If2: 10.0.0.4, Main2: 10.0.0.4 :: Time: Fri Sep 19 11:27:07 2003
Add MPR. Main: 10.0.0.6
Add MPR. Main: 10.0.0.2
New neighb on 2 hop. If: 10.0.0.2, Main: 10.0.0.2
                     If2: 10.0.0.1, Main2: 10.0.0.1 :: Time: Fri Sep 19 11:27:07 2003
Add MPR. Main: 10.0.0.6
Add MPR. Main: 10.0.0.2
New neighb on 2 hop. If: 10.0.0.2, Main: 10.0.0.2
                     If2: 10.0.0.5, Main2: 10.0.0.5 :: Time: Fri Sep 19 11:27:07 2003
Add MPR. Main: 10.0.0.6
Add MPR. Main: 10.0.0.2
Add MPR Sel. If: 10.0.0.2, Main: 10.0.0.2
...
```

Appendix F
Configuring *iPAQ 3950*

In order to integrate our Ad Hoc protocol into the testing platform, we need to configure the GNU/Linux system in the iPAQ 3950 [5] nodes. The system has to be configured for supporting Wireless LAN, audio and our *AD Hoc protocols implementation.* The following sections present a detailed sequence of steps for a successful configuration of the nodes.

Basic steps are the following:

1. Install Linux.
2. Configure wireless LAN.
3. Configure iPAQ for access to the Internet.
4. Install AODV protocol.
5. Configure sound in iPAQ.

Installation of *GNU/Linux* operating system

First of all, installation of an operating system depends directly on the harware supported. In our case, the hardware is very specific and it puts a lot of restrictions in order to choose a Linux distribution.

This is the iPAQ hardware specification:

- **CPU**. Intel XScale-PXA250 400 MHz revision 4.
- **RAM**. 64 MB.
- **ROM**. 32 MB Flash ROM.
- **Sound**. Philips UDA1380.
- **Wireless card**. PCMCIA D-Link DCF-660W.

With this hardware configuration, we chose the *Familiar* Linux distribution. We found this distribution, applications and documentation in [1].

Once chosen a distribution, the steps to install Linux are:

1. Connect the iPAQ via the USB cradle to Win2K machine.
2. Use the ActiveSync application to connect the iPAQ from the PC.
3. Copy boot blaster program (we chose it in [2]) to the default folder on the iPAQ from the Windows machine using drag and drop or cut and paste. Ignore any messages that say it *may need to convert* file formats.
4. Copy boot loader program (we chose it in [3]) to the default folder on the iPAQ from the Windows machine. Again, ignore any messages that say it *may need to convert* file formats.
5. On the iPAQ, find *BootBlaster3900.exe* and then execute it.
6. From the *Flash* menu, select *Save.* This will save a copy of the current bootloader to DRAM on the iPAQ (under the name *saved_bootldr.bin*).
7. Copy the *saved_bootldr.bin* off of the iPAQ and put it in a safe place in order to be able to restore it.
8. From the *Flash menu*, select *Save Windows gz.* This will copy and compress all the flash ROM on your iPAQ into a *.gz file* along with a file containing the asset information from your iPAQ. This will take a while. After it is complete, copy these files to the PC to save them. Under normal circumstances, installing Linux will not touch the asset partition in flash, but it is safer to have a backup copy.
9. From the *Flash* menu on BootBlaster, select *Program.* A file dialog will open allowing you to select the bootloader to use. Select the bootloader from step 5). This step can take a while.
10. From the *Flash* menu on BootBlaster, select *Verify.* If it does not say that you have a valid bootloader, do **NOT** reset your iPAQ. Instead, try programming the flash again. If that doesn't work, program your flash with your saved bootloader.
11. Configure the terminal emulator properly with:
    - 115200 baud.
    - 8N1.
    - No flow control.

- No hardware handshaking.

12. To get to the bootldr's command line prompt (boot¿) and avoid booting Windows CE: depress and hold the center of the joypad while pushing the recessed reset button.

13. Unlock flash with this command: `pflash 0x40000 0xffff 0` by typing it at the boot prompt.

14. Reboot iPAQ again type `reboot` for example.

15. Type `partition reset`.

16. Type `load root`

17. Send the file system image using *Xmodem* with the terminal emulator. We have used [4] file system image. This file contains *2.4.19-rmk4-pxa2-hh8* kernel which has all modules necessary for configuring sound and wLAN (This step consumes 1 hour approximately).

18. Type `boot`.

Now, iPAQ has a Linux operating system running.

Configure Wireless LAN

Wireless configuration is not very hard following these steps:

1. Find out the pcmcia module with the command `cardctl ident`. In our case, we obteined the following results:

```
Socket 0:
product info: "D-Link", "DCF-660W", ""
manfid: 0xd601, 0x0005
function: 6 (network)
```

2. Add the following lines to `/etc/pcmcia/config`:

```
card "D-Link DCF-660W"
  manfid 0xd601, 0x0005
  bind "orinoco_cs"
```

3. Edit `/etc/pcmcia/network.opts` as the following example:

```
# Network adapter configuration
#
# The address format is "scheme,socket,instance,hwaddr".
#
# Note: the "network address" here is NOT the same as the IP
# address.
# See the Networking HOWTO.  In short, the network address is
# the IP address masked by the netmask.
#
case "$ADDRESS" in
*,*,*,*)
INFO="Sample private network setup"
#$ Transceiver selection, for some cards -- see 'man ifport'
IF_PORT="10base2"
# Use BOOTP (via /sbin/bootpc, or /sbin/pump)? [y/n]
BOOTP="n"
#Use DHCP(via /sbin/dhcpcd,/sbin/dhclient,or /sbin/pump)?[y/n]
#DHCP="y"
#PUMP='n'
#If you need to explicitly specify a hostname for DHCP requests
#DHCP_HOSTNAME=""
#Host's IP address, netmask, network address, broadcast address
IPADDR="10.0.0.3"
NETMASK="255.255.255.255"
NETWORK="10.0.0.0"
BROADCAST="255.255.255.255"
# Gateway address for static routing
GATEWAY="10.0.0.1"
# Things to add to /etc/resolv.conf for this interface
DOMAIN="netlab.org"
SEARCH=""
DNS_1=""
DNS_2=""
DNS_3=""
# NFS mounts, should be listed in /etc/fstab
MOUNTS=""
# If you need to override the interface's MTU...
MTU=""
# For IPX interfaces, the frame type and network number
IPX_FRAME=""
IPX_NETNUM=""
# Extra stuff to do after setting up the interface
start_fn () { return; }
# Extra stuff to do before shutting down the interface
```

```
stop_fn () { return; }
# Card eject policy options
NO_CHECK=n
NO_FUSER=n
;;
esac
```

4. Edit '/etc/pcmcia/wireless.opts as follows:

```
case "$ADDRESS" in

#$NOTE: Remove the following six lines to activate the samples below...
# --------- START SECTION TO REMOVE -----------
*,*,*,*)
ESSID="netlab"
MODE="ad-hoc"
;;
# ---------- END SECTION TO REMOVE ------------
```

Configure iPAQ to access the Internet

There are two differents ways of configuring iPAQ for accessing the Internet, by a *ppp* connection or by a *USB* connection.

iPAQ 3950 has many problems with USB connection so it is easier configuring a ppp conection.

Steps for configuring ppp:

1. Make sure that `/etc/passwd` contains a line like.

   ```
   ppp::101:101:PPP User:/home/ppp:/usr/sbin/pppd
   ```

2. Create or modify `/etc/ppp/options` as follows.

   ```
   -detach
   defaultroute
   noauth
   local
   nocrtscts
   lock
   ```

```
lcp-echo-interval 5
lcp-echo-failure 3
/dev/tts/0
115200
```

3. Make sure `/etc/modules.conf` has the appropriate aliases.

```
alias /dev/ppp          ppp_generic
```

4. Make sure `/usr/sbin/pppd` is executable by user ppp.

```
chmod 4755 /usr/sbin/pppd
```

5. Load modules.

```
insmod slhc.o
insmod ppp_generic.o
insmod ppp_async.o
```

iPAQ is now configured. To access iPAQ from desktop just type this command:

```
pppd /dev/ttyS1 115200 192.168.0.1:192.168.0.2 debug nodetach
local noauth nocrtscts lock user ppp connect "/usr/sbin/chat -v
-t3 ogin--ogin: ppp"
```

Establishing a connection with iPAQ can require several attempts.

To access to iNET it is neccessary to copy `/etc/resolv.conf` as following.

```
scp /etc/resolv.conf root@192.168.0.2:/etc/resolv.conf
```

Finally, *masquerading* must be available for communicate the iPAQ to the outside world.

```
iptables --flush
iptables --table nat --flush
iptables --delete-chain
iptables --table nat --delete-chain
iptables --table nat --append POSTROUTING --out-interface eth0 -j
   MASQUERADE
iptables --append FORWARD --in-interface ppp0 -j ACCEPT
```

Now iPAQ is able to surf in the Internet.

Install AODV protocol

For installing AODV we need a cross compiler and a compile kernel. We have used *ToolChain cross compiler* [6] in order to compile kernel and aodv code for iPAQ.

This version of ToolChain is the one (called *monmotha*) that can compile code for the xscale processor. It must be installed in `/opt/arm`.

We have used *2.4.19-rmk4-pxa2-hh8* kernel source code. We have used CVS environment to download. These are the steps:

1. `export CVSROOT=:pserver:anoncvs@handhelds.org:/cvs`
2. `cvs login`. Use `anoncvs` when password was asked.
3. `cvs checkout -r K2-4-19-rmk4-pxa2-hh8`. Do this in at `/opt/src`.

When we have downloaded the kernel source and installed the cross compiler, we can compile the kernel for iPAQ.

We are now at `/opt/src/linux/kernel` and we have to follow next steps:

1. Edit configure file at `./arch/arm/def-configs/h3900` and enable netfilter module by `NF_QUEUE=m`.
2. `make h3900_config`
3. `make oldconfig`
4. `make dep`
5. `make zImage`
6. `modules`
7. `mv System.map System.map.orig`
8. `mv scripts/ipkg-make-kernel-packages scripts/ipkg-make-kernel-packages.orig`
9. Download a new version of this script from [7].
10. Download a *ipkg-build* from [8].
11. `mkdir ipkgs; cd ipkgs`
12. `../scripts/ipkg-make-kernel-packages $PWD/.. 2.4.19-rmk4-pxa2-hh8`

13. `ln -s /opt/src/linux/kernel/include/asm /opt/arm/arm-linux/include/asm`

14. `ln -s /opt/src/linux/kernel/include/linux /opt/arm/arm-linux/include/linux`

We had some problems with *i2c* devices, so, before `make dep` we have done `make xconfig` and we have disabled these devices.

Now we have a compiled kernel and its modules and we have to compile other programs.

In order to install aodv, we need to install the new *ip_tables* module which we have compiled previously. One of the all possibles ways to install is the following:

1. We guess that PC and iPAQ is already connected by a PPP conection.
2. `scp iptables-modules-2.4.19-rmk4-pxa2-hh8__arm.ipk root@192.168.0.2:/`
3. `ssh root@192.168.0.2`
4. `cd /ipkg`
5. `install iptables-modules-2.4.19-rmk4-pxa2-hh8__arm.ipk -force-depends`
6. `modprobe ip_filter`

Now we can compile and install properly the AODV code. In this iPAQ we have to do some modifications in the AODV original code.

1. Update this line at *Rules.make*:
   `KINC_ARM=-I/opt/src/linux/kernel/include`.
2. Add this line at *Rules.make*:
   `INCLUDE=-I/opt/arm/arm-linux/include`.
3. Comment structure *in_pktinfo* sited at file *aodv_socket.c*.
4. `make arm`

Aodv is already for using in iPAQ, the installation process is the following:

1. We assume that PC and iPAQ are already connected by a PPP conection.
2. We assume that we are at the main directory of aodv.

3. `scp kernel/ipq-arm.o root@192.168.0.2:/lib/modules /2.4.19-rmk4-pxa2-hh8/kernel/net/ipv4/ipq.o`
4. `scp kernel/rl-arm.o root@192.168.0.2:/lib/modules/ 2.4.19-rmk4-pxa2-hh8/kernel/net/ipv4/rl.o`
5. `scp uu-daemon/aodvd-arm root@192.168.0.2:\~`
6. `scp uu-daemon/aodv.sh root@192.168.0.2:\~`

Now AODV is ready for using.

Configure sound in iPAQ

It is easy to configure sound in iPAQ, although the order in loading modules is very important because iPAQ could be frozen.

This is the correct order:

1. `insmod uda1380.o`
2. `insmod h3900-uda1380.o`

Sound is enabled.