Bertrand Grandgeorge
SSC 3

SEMESTER PROJECT

Report

"AODV routing algorithm for multihop Ad Hoc networks"

PROJECT ADVISOR
David CAVIN
Distributed Systems Lab.
EPFL-IC-IIF-LSR

PROJECT CO-ADVISOR
Yoav SASSON
Distributed Systems Lab.
EPFL-IC-IIF-LSR

PROJECT DIRECTOR
Prof. André SCHIPER
Distributed Systems Lab.
EPFL-IC-IIF-LSR

Table of Contents

1. The project

Intitulé du projet:

« Le standard IEEE 802.11 qui décrit le fonctionnement des réseaux sans fil ne considère que des environnements single-hop dans lesquels chaque station n'est capable de communiquer directement qu'avec son entourage. Si on considère des réseaux de plus grande taille sans infrastructure cablée, il est nécessaire d'envisager un algorithme qui permette de propager un message en passant par plusieurs stations relais (sortes de router). Pour résoudre ce problème, il existe déjà plusieurs algorithmes de routage bien adaptés au contexte des réseaux ad hoc : AODV (Ad hoc On-demand Distance Vector).

Le premier but de ce projet consiste à porter une implémentation JAVA ou C de l'algorithme MAODV Multicast Ad hoc On-demand Distance Vector), dérivé de AODV qui permet de faire du multicast. Dans un deuxième temps, une simple application illustrant son fonctionnement sera codée et déployée sur les iPaq du réseau mobile du LSR. »

Translation :

The IEEE 802.11 standard which describes wireless functioning takes only in account single-hop environments in which each station is able to communicate directly only with its neighbors. If we consider bigger networks without wired infrastructure, it is necessary to consider an algorithm which permits the propagation of a message through several relay stations (kind of routers). In order to solve this problem, several routing algorithms adapted to Ad Hoc networks already exist: AODV (Ad hoc On-demand Distance Vector).

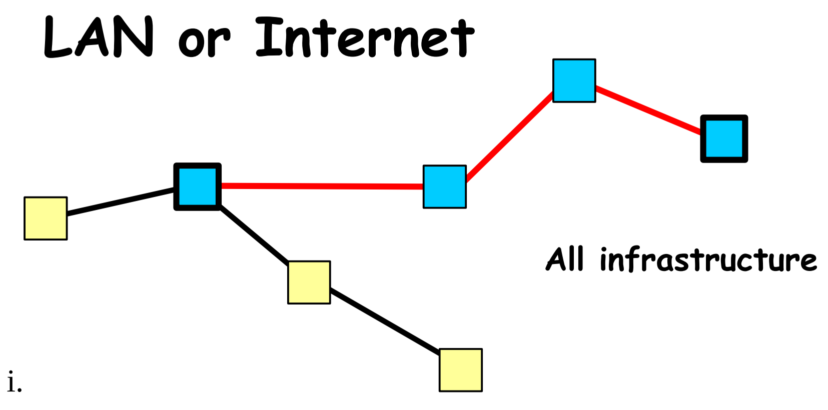
The first goal of this project consists in porting a JAVA or C implementation of the MAODV algorithm (Multicast Ad hoc On-demand Distance Vector), derived from AODV which permits multicast. In a secondary part, a simple application illustrating its use will be coded on iPaqs from the LSR's mobile network.

2. Introduction to Ad-Hoc Networks

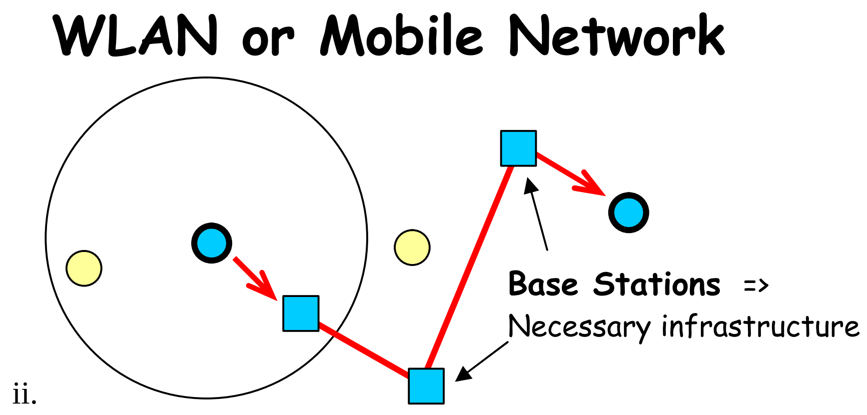
2.1. Overview

Ad-Hoc networks are the new deal in wireless communication. Unlike traditional networks, no infrastructure is actually needed. All "nodes" (i.e. communicating systems) are equal in role, as no client or server exists. In addition to traditional wireless networks problems such as, bandwidth optimization, power control and transmission quality enhancement, Ad-Hoc networks brings his load of new challenges. The actual lack of infrastructure requires the introduction of new tasks like discovery and maintenance, addressing (no server to give an address to each node!) or routing.

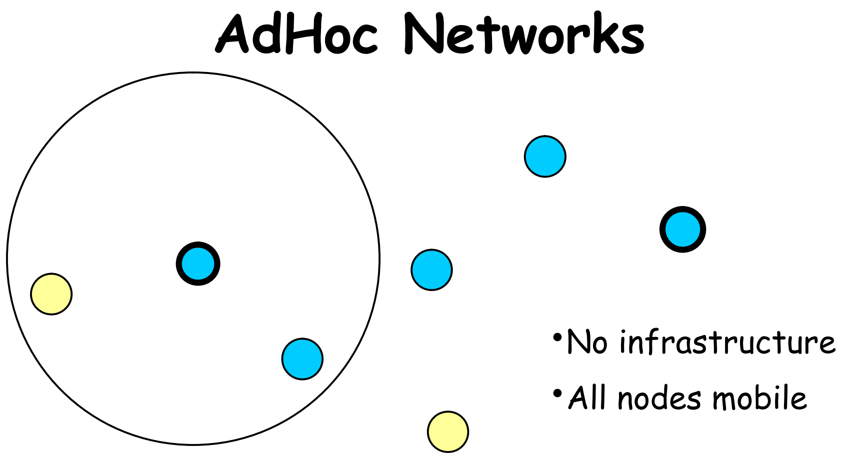
From wired networks to AdHoc…



i.



ii.



iii.

2.2. Characteristics and issues

Nodes are *equal*: all nodes can communicate with each other, with the same priority, regardless of the position. This means that if two nodes are out of reach and want to communicate, they have first to find each other over the network. But as no central control exists (server, router), Ad-Hoc cannot rely on IP-like address which would uniquely identify a node. So the first challenge is to provide some kind of identifier of each node, for it to know how to “call” one another. Secondly, as all should be reachable, distant (not in the same cell) nodes should communicate through a path of other nodes, acting as routers. This is another subject of research: routing with nodes as routers!

Frequent *changes in topology*: as all nodes are mobile, no topology of the network can be guaranteed. The challenge here will be to try to predict topology changes, in order to ascertain permanent connectivity.

Wireless has *lower capacity* that wired: this is a technical issue, for now, wireless links are able to transfer less data than wired links.

Security is limited: as no physical connection is needed, anyone can connect to any wireless network. One should be aware that wireless communication will continue to be less secure than wired.

Higher *loss rates* and *delays*: another technical issue is the link itself: air. Electromagnetic waves encounter many interferences coming from all environment and causing higher loss rates and delays.

Rely on *battery*: mobility requires the system to be transportable and therefore to function with a battery. This becomes an issue if it wears out, by breaking the connectivity of surrounding neighbors.

Power limited: each node even if equal in role can be different, and more specifically have different power (CPU, memory …). This adds complexity in determining the speed of the transfer.

Limited storage space: few data can be stored at a time.

2.3. Terminology

Node: any system (Portable computer, pocket computer, or even cellular phone) that is part of the wireless network.

Cell: abstract field of reception of a particular node. A node can send or receive only to nodes within the cell.

Packet: data unit of message. A message (data) is sent in several packets.

Source node: the node that is willing to send a message.

Destination node: the intended target of the message.

Forwarding node: all nodes between source and destination nodes. They are expected to forward messages from source to destination.

Hop: path from one node to another. A route contains one more hop than forwarding nodes.

Broadcast: the only to transfer data in wireless network ; the signal can be received from anywhere in the emission area.

Unicast: opposed to broadcast ; a message is sent to one destination only.

Flooding: the simplest routing algorithm. The data is just sent to everybody and forwarded, until the destination is attained. Potentially all the network can receive the data, resulting in saturation (congestion).

Congestion: when too much data is sent on a network link or zone, errors due to collisions and physical limits may appear, resulting in data loss or delays.

3. A word on 802.11 and iPacks

802.11 are the IEEE Official standards for wireless communication:

- IEEE 802.11-1997:
  Wireless LAN medium access control and physical layer specifications

- IEEE 802.11a-1999:
  High-speed physical layer

- IEEE 802.11b-1999:
  Higher-speed physical layer extension

- IEEE 802.11d-2001:
  Specification for operation in additional regulatory domains

802.11 works with two different operating modes with different architectures:

- *Infrastructure mode*: Cooperative and structured (WLAN).
- *Independent (ad Hoc) mode*: Concurrent and distributed.

The iPacks are Compaq's solution of pocket PCs. It can receive a 802.11 network card through a PCMCIA port.

iPaq 3660

Operating system
Windows® Powered Pocket PC
Linux

Processor
206 MHz Intel StrongARM 32-bit RISC

Memory
32 MB RAM memory
16 MB ROM memory

Development environment
Visual C++
Java

Porting JAVA applications on these machines requires caution as the version supported is only v1.6.1. This is not a final statement, considering the evolution of such equipment.

4. Routing in MANET networks

The goal is to find stable routes (despite mobility) to rely on for packets dissemination. Furthermore the route should be—if not optimized—short. One problem in MANET Networks is, as discussed before, the lack of Identifier for each node (IP-like address). Supposing that this problem is solved by for instance, a number reflecting a unique serial number of the system, we will now concentrate on how to send a message to an intended target.

4.1. Proactive/reactive protocols

Routing protocols are divided in two categories: *proactive* and *reactive* protocols. Some attempts have been made to develop adaptive/hybrid protocols able to work well on all environments.

Proactive protocols:
- Always maintain routes
- Little or no delay for route determination
- Consume bandwidth to keep routes up-to-date
- Maintain routes which may never be used

Reactive protocols:
- Lower overhead since routes are determined *on demand*

- Significant delay in route determination
- Employ flooding (global search)
- Traffic control may be congestive

Hybrid protocols (combination of proactive and reactive) are also in research.

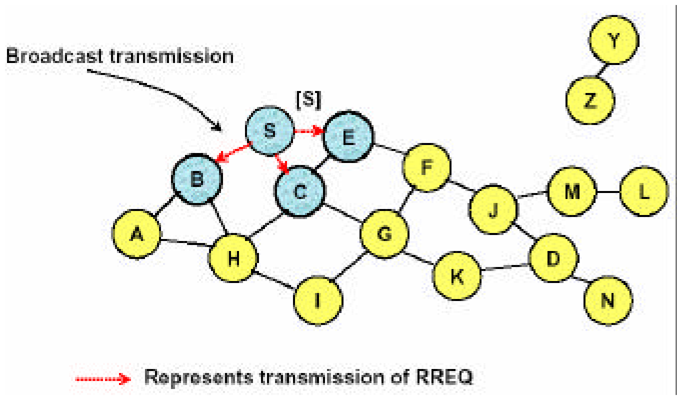
4.2. Examples

DSDV (Destination Sequenced Distance Vector: proactive): Each node maintains a routing table which stores the next hop, cost metric towards each destination and a sequence number that is created by the destination itself. Each node periodically forwards routing table to neighbors. Each node increments and appends its sequence number when sending its local routing table. Each route is tagged with a sequence number; routes with greater sequence numbers are preferred. Each node advertises a monotonically increasing even sequence number for itself. When a node decides that a route is broken, it increments the sequence number of the route and advertises it with infinite metric
Destination advertises new sequence number

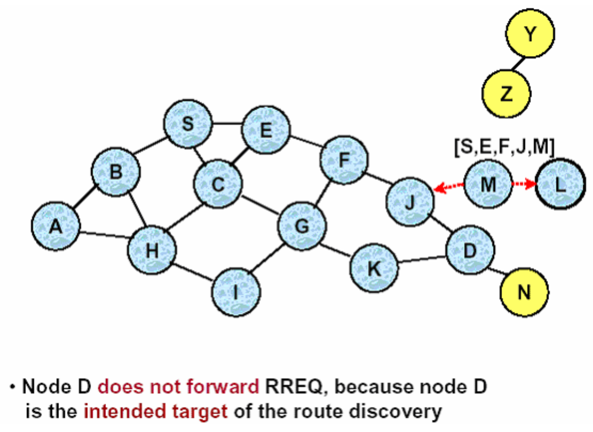
DSR (Dynamic Source Routing: reactive): The idea is that when a source node wants to send a packet to a destination, but does not know a route to it, the source initiates a *route discovery*.

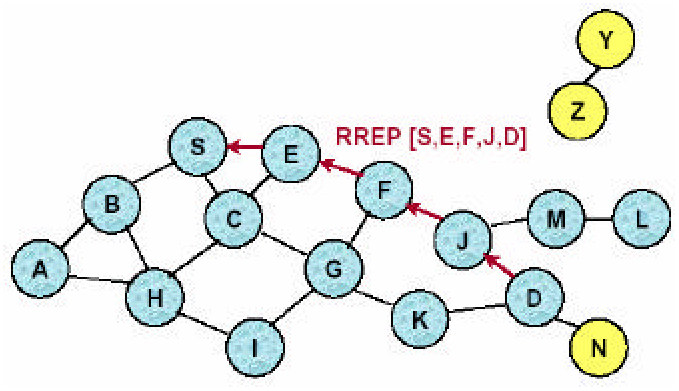
By sending a *route request* (RReq), the source node floods the network, in order to let all nodes know who (what destination node) it is looking for. The RReq stores the route taken by appending the identifier of each forwarding nodes and propagates in the network until the intended target (destination) receives it.

This destination node sends then a *route reply* (RRep), using the *reversed route* taken by the RReq. The RRep includes the complete route from source to destination. On reception of the RRep, the source can finally send its packet after including the entire route in the packet header. Intermediate nodes use the *source route* included in the packet to determine to which node the packet should be forwarded.

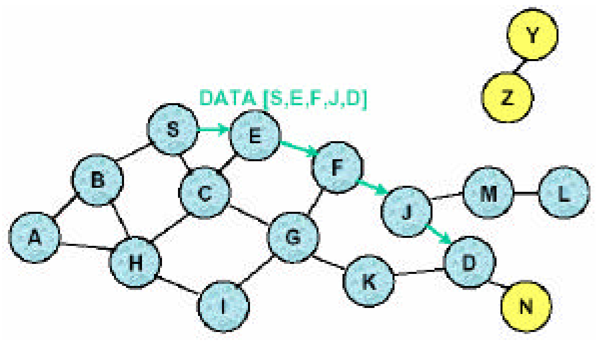


Source S broadcasts a RReq in the network which only forwarded only one time per node.


[S,E,F,J,M]
S
E
B
C
A
H
I
G
F
J
K
M
L
D
N
Y
Z
• Node D does not forward RREQ, because node D is the intended target of the route discovery



Intended target D sends the route reply.



The source S sends the data packet with the route included in the header.

Other names like "*Optimized Link State Routing*" (OLSR: proactive) or "*Zone Routing Protocol*" (ZRP: Hybrid), also exist.

AODV (Ad hoc On-demand Distance Vector: reactive): improvement of DSR, see next chapter.

5. AODV

As seen before, in DSR sources includes routes in packet header. Depending on the length of the route, this header can become heavy (even if the content of the packet is small) and degrade performances. AODV solves this problem by improving DSR in that it does not need to include entire route in each packet.

5.1. Overview

AODV stands for AdHoc On-Demand Distance Vector and is therefore (on-demand) a reactive protocol. It could actually be seen as a hybrid protocol as it has also some proactive characteristics (route table at each node).

The first change from DSR to AODV is the introduction of routing tables at the nodes, so that packets do not have to contain entire route in their header. But AODV retains the desirable feature of DSR, that routes are maintained only between nodes which need to communicate (active routes).

Route Requests are forwarded in same way of DSR.
When a node re-broadcasts (forwards) a Route Request, it sets up a reverse path pointing towards the source of the RReq.
The intended destination replies by sending a Route Reply (RRep) which is unicasted to next hop in the newly built reverse route, to reach the originator of the RReq.
On reception of each control message (RReq, RRep…) a node can update its routing table in order to take in account evolution of the network (i.e. topology changes, obsolete routes…).

5.2. AODV Terminology

(From the IETF's "manet-aodv-12" draft)

active route:
A route towards a destination that has a routing table entry that is marked as valid. Only active routes can be used to forward data packets.

broadcast:
Broadcasting means transmitting to the IP Limited Broadcast address, 255.255.255.255. A broadcast packet may not be blindly forwarded, but broadcasting is useful to enable dissemination of AODV messages throughout the ad hoc network.

destination:
An IP address to which data packets are to be transmitted. Same as "destination node". A node knows it is the destination node for a data packet when its address appears in the appropriate field of the IP header. Routes for destination nodes are supplied by action of the AODV protocol, which carries the IP address of the destination node in route discovery messages.

forwarding node:
A node that agrees to forward packets destined for another node, by retransmitting them to a next hop that is closer to the unicast destination along a path that has been set up using routing control messages.

forward route:
A route set up to send data packets from a node originating a Route Discovery operation towards its desired destination.

invalid route:
A route that has expired, denoted by a state of invalid in the routing table. An invalid route is used to store the previously valid route information for an extended period of time. An invalid route may not be used to forward data packets.

originating node:
A node that initiates an AODV message to be processed and possibly retransmitted by other nodes in the ad hoc network. For instance, the node initiating a Route Discovery process and broadcasting the RREQ message is called the originating node of the RREQ message.

reverse route:
A route set up to forward a reply (RREP) packet back to the originator from the destination or from an intermediate node having a route to the destination.

sequence number:
An increasing number maintained by each originating node. When used in control messages it is used by other nodes to determine the freshness of the information contained from the originating node.

valid route:
See active route.

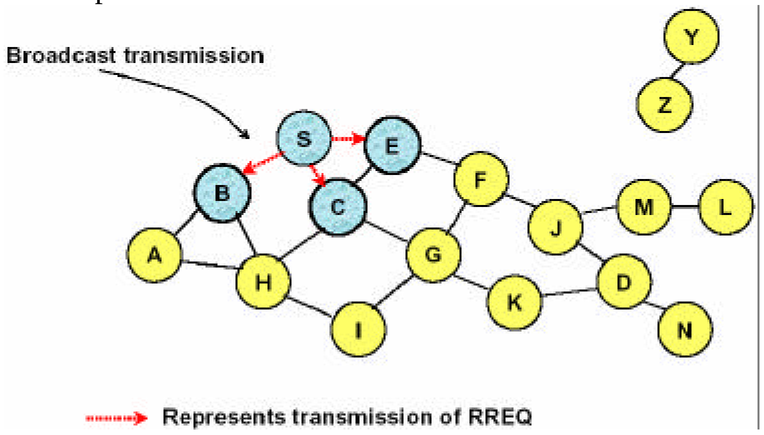
5.3. Functioning principle

AODV uses 5 different types of control messages: Route Requests (RReq), Route Reply (RRep), Route Reply Acknowledgement (RRepAck), Route Error (RErr), and Hello (Hello) messages.

We can highlight some steps in the algorithm, basically for each type of message sent:

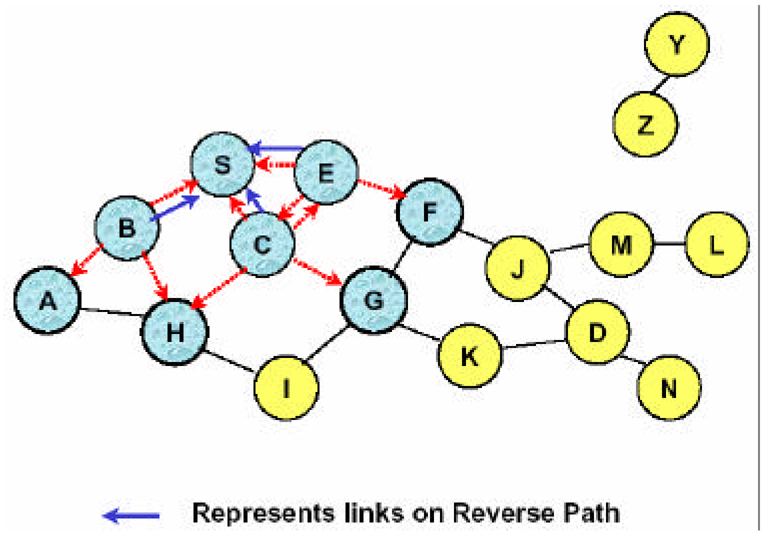
5.3.1. RReq broadcast

Before sending a message to a destination, the originating node consults its routing table to look if a next Hop (next forwarding node in route) exists. If none is found or route is obsolete, the node need to initiate a route discovery.
So, like DSR, the originating node broadcasts a RReq in the network. Each RReq contains information on the originating and destination node (ID and Sequence number). It is also supposed to count the hops from source to destination.



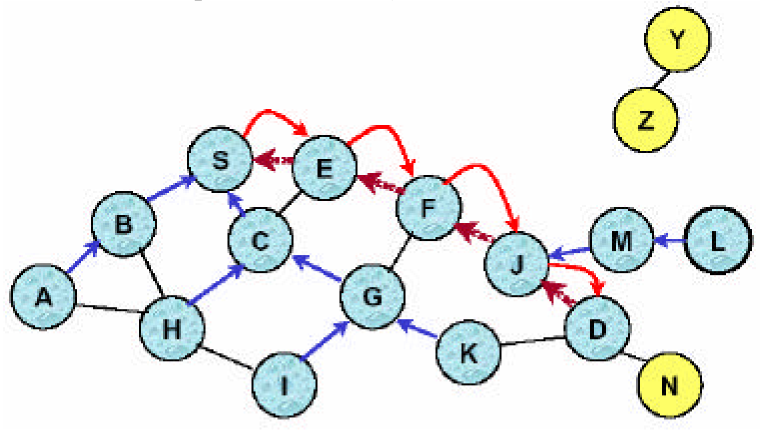
5.3.2. RReq forward

All node receiving a RReq forwards it but only one time (as it will receive several time the same RReq from its neighbors). At this time the route to the previous node is also created updated. If a node is the destination itself, or has an active route to destination, ti responds by a RRep.



5.3.3. RRep generation

RRep can be generated either from a destination node, or a forwarding node. In both cases, they naturally don't forward the RReq, but the information included in RRep is slightly different. RRep are *unicasted* to previous node (the one from which the RReq comes).



5.3.4. RRep forwarding

Each node on route receives a RRep that it needs to forward to next hop found by consulting its table for the originating node. The RReps are *unicasted* so that only nodes on route are supposed to forward them to the originating node. Naturally, these forwarding nodes also update their table with the information contained in the RRep, for the forward route to destination.

5.3.5. Error detection

A RErr message is iteratively unicasted to all *precursors* (list of forwarding nodes in a route) stored in node's routing table. A node initiates a RErr message if either:

- it detects a link break for the next hop of an active route in its routing table while transmitting data, or
- it gets a data packet destined to a node for which it does not have an active route, or
- it receives a Rerr from a neighbor for one or more active routes.

5.3.6. Table maintenance

A route is only updated if the new sequence number coming from the AODV control message is either:

- higher than the destination sequence number in the route table, or
- the sequence numbers are equal, but the hop count (of the new information) plus one, is smaller than the existing hop count in the routing table, or
- the sequence number is unknown.

5.4. JAVA Implementation

Note: I tried as much as I could to obey the IETF draft for manet-aodv 12th version[1], but integration cannot be exactly the same as described for technical reasons (mainly framework integration). I had a C implementation from Uppsala University, but considering the fact that I had to integrate my work in a framework being built by another group, I decided it was simpler to start right from the scratch. This way I also avoided the problem of translation in a language that might not have all the functionality of C.

5.4.1. Message Types

Each message type becomes a JAVA object with fields described as below. One difference with the draft is that message types are on 16bits (char) instead of 8. This is to be compatible with the rest of the framework that includes a configuration file for this kind of constants. But I still included the type as a field for the message objects. Another one is the length of the ID (IP in the draft) which is 64bits instead of 32, for the same reason of compatibility.

Fields of each message type: (Different from draft)

Route Request message (RReq):

```
 0                   1                   2                   3 .
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Type              |     Flags     |   Hop Count   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            RREQ ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination ID Address                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination ID Address                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Destination Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Originator ID Address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Originator ID Address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Originator Sequence Number                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type: type of message (set to 1 in the draft)

Flags: 5 RReq parameters: J (join flag), R (repair flag) both reserved for multicast, G(Gratuitous RRep flag): indicates whether a gratuitous RRep should be unicast to the node specified in the Destination ID field.
D (Destination only flat): indicates only the destination may respond to this RReq.
U (Unknown sequence number)

Hop_Count: increasing value to count the hops from source to destination

RReq_ID: A sequence number uniquely identifying the particular RReq when taken in conjunction with the originating node's ID.

Dest_ID: The Identifier (address) of the destination for which a route is desired.

Dest_SeqNum: The greatest sequence number received in the pas by the originator for any route towards the destination.

Orig_ID: The Identifier of the node which originated the Route Request.

Orig_SeqNum: The current sequence number to be used for route entries pointing to (and generated by) the originator of the route request.

Route Reply message (RRep):

```
 0                   1                   2                   3 .
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+                                             .
|   Prefix size     |                                             .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type              |     Flags      |   Hop Count   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            RREQ ID                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination ID Address                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination ID Address                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Destination Sequence Number                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Originator ID Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Originator ID Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Lifetime                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type: type of message (set to 2 in the draft)

Flags: R(Repair flag), A(Acknowledgement field)

Prefix_Size: If nonzero, the 5-bit Prefix Size specified that the indicated next hop may be used for any nodes with the same routing prefix (as defined by the Prefix Size) as the requested destination.

Hop_Count: The number of hops from the Originator ID address to the Destination ID address. For multicast route requests this indicates the number of hop to the mulicast tree member sending the RRep.

Dest_ID: Identifier (address) of the destination for which a route is supplied.

Dest_SeqNum: The destination sequence number associated to the route.

Orig_ID: Identifier of the node which originated the RReq for which the route is supplied.

Lifetime: The time in milliseconds for which nodes receiving th RRep consider the route to be valid.

Route Reply Acknowledgment message (RRepAck):

```
 0                   1         .
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Type              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type: type of message (set to 4 in the draft)

Route Error message (RErr):

```
 0                   1                   2                   3 .
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Type              |     Flags     |   DestCount   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            RREQ ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Unreachable Destination ID Address               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Unreachable Destination ID Address               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Unreachable Destination Sequence Number            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Additional Unreachable Destination ID Address         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Additional Unreachable Destination ID Address         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Originator Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type: type of message (set to 3 in the draft)

N_flag: No delete flag; set when a node has performed a local repair of a link, and upstream nodes should not delete the route.

Unreacheable_Dest_SeqNum: The sequence number in the route table entry for the destination listed in the previous Unreachable Destination ID address field.

Dest_ID: The ID address of the destination that has become unreachable due to a link brake.

Dest_Count: The number of unreqchable destinations included in the message; MUST be at least 1.

Dest_ID and Unreacheable_Dest_SeqNum are actually arrays containing the addresses and Sequence Numbers.

5.4.2. Route table entries and configuration parameters

Route entries are specified by the AODV draft, but not all of them are used in this implementation (though present) they are:

- Destination ID
- Destination Sequence Number
- Valid Destination Sequence Number
- Interface: *not used here*
- Hop Count: number of hops needed to reach destination
- Next Hop: next node ID to which message should be sent
- List of Precursors: list of forwarding nodes in the route
- Lifetime: expiration or deletion **time** of the route
- Routing Flags: *not used here*
- State: *not used here*

Parameters: (in configuration file with default value), again all of them are present in this implementation, but not all used. They are rather explicit.

AODV Parameters (as specified in draft):

Parameter Name	***Default Value (update formula)***
ACTIVE_ROUTE_TIMEOUT	3,000 Milliseconds
ALLOWED_HELLO_LOSS	2
BLACKLIST_TIMEOUT	RREQ_RETRIES * NET_TRAVERSAL_TIME
DELETE_PERIOD	
HELLO_INTERVAL	1,000 Milliseconds
LOCAL_ADD_TTL	2
MAX_REPAIR_TTL	0.3 * NET_DIAMETER
MIN_REPAIR_TTL	
MY_ROUTE_TIMEOUT	2 * ACTIVE_ROUTE_TIMEOUT
NET_DIAMETER	35
NET_TRAVERSAL_TIME	2 * NODE_TRAVERSAL_TIME * NET_DIAMETER
NEXT_HOP_WAIT	NODE_TRAVERSAL_TIME + 10
NODE_TRAVERSAL_TIME	40
PATH_DISCOVERY_TIME	2 * NET_TRAVERSAL_TIME
RERR_RATELIMIT	10
RING_TRAVERSAL_TIME	2*NODE_TRAVERSAL_TIME*(TTL_VALUE+TIMEOUT_BUFFER)
RREQ_RETRIES	2
RREQ_RATELIMIT	10
TIMEOUT_BUFFER	2
TTL_START	1
TTL_INCREMENT	2
TTL_THRESHOLD	7
TTL_VALUE	

5.4.3. Integration in Framework

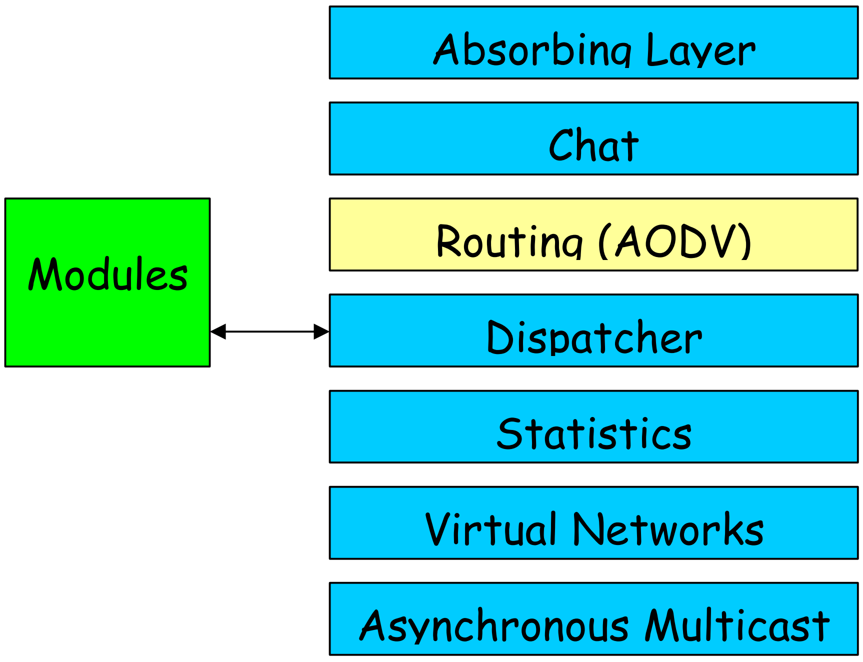
The difficulty of this project resided in its integration in a changing framework. It was necessary to have a cooperation with the framework implementers in order that they provide enough tools for an algorithm like AODV to be implemented. However, working on integrating something in a changing framework delays the implementation.

The framework includes a "*message pool*" to avoid building too many objects which is costly in JAVA. To use it, one has to first get access to this message pool, and to implement a message factory for each message object. The message factory actually tells the pool, how to construct the message. Then to build a message, we ask the message pool to return a message object of a specified type. And finally the message can be set with parameters, and at last sent.

The framework is composed of layers (Asynchronous Layers) arranged in a stack as a network application requires. Each layer has a *thread* and a *buffer* in which messages are stored to be consumed by upper layer. A layer *notifies* the upper layer if a message is put in its buffer, and is *awaken* by the lower layer on reception of a message (which is buffered in the lower layer).

AODV is a complete layer (*routing*) and takes its place above the dispatcher.

See figure next page.



The *dispatcher* is a layer dedicated to deliver message to the proper layer or module.

The *modules* are side programs (not part of the layer stack), which offer a special feature useable by several layers. Already implemented is the Hello Module.

Application layer is situated on top of the stack. The implemented application is a chat working in multihop.

Virtual Networks is situated above the communication layer (Asynchronous Multicast) and its purpose is to create virtual networks so that only nodes on the same network can directly communicate with each other. It was implemented so that multihop communication could be tested.

5.4.4. Classes and Methods

AODV package contains 11 classes: 4 for the message objects and their corresponding message factories, one for the route table, one for its entries, and one for the algorithm. There are only four messages because Hello message were implemented in Module on side of the layer stack.

Methods description (in class “AODV”):
Apart from the methods for package integration in framework (Constructor, Initialize, Send, sendMessage, handleMessage, Startup), the methods are in concordance with AODV steps.

Constructor: empty, but necessary for building the stack.

Initialize: all the initializations needed, i.e. access to the pool, reading of the configuration file (manet.config) for specific parameters…

sendMessage: overrides super-class sendMessage method in order to handle message coming from the above layer. Actually calls this class’s handleMessage.

Send: called when sending a message from the class, this method sends the actual message of control and throws an error message stating the nature of the message for which the error occurred.

Startup: start the thread inherited from Asynchronous Layer. It will sleep after sending a message, and will be awaken by the lower layer upon message arrival.

handleMessage: takes action depending on the message received.

```
If (AODV message type) process message with proper process<Msg> method
Else if (destination is thisNode) put message in buffer
Else if (destination is 0) put message in buffer and broadcast again if (TTL>1)
Else if (route for destination known) send to next forwarding node
Else generate a route discovery for destination (send an RReq).
```

RReqGen: Asks a route for a message to reach a destination. The RReq is buffered (to avoid receiving duplicates from neighbors) and then broadcasted. RReq sending rate is limited. Repeat RReq attempts for a route discovery of a single destination is not yet implemented. It must use a binary exponential back off (to set waiting time before resending an RReq). An expanding ring search technique should also be used to prevent unnecessary flooding of RReq.

RReqProcess: takes action upon RReq arrival. Creates or updates a route to previous hop and to Originator. If node is destination, generates a Route Reply. If not, forward RReq.

RReqFwd: Forwards (broadcast) Route Request if node is not destination and RReq's TTL is greater than 1.

RRepGen: Route Reply generation from destination. The destination sends a route reply to the previous node.

RRepGenInter: Route Reply generation from intermediate node knowing a route to destination. The node update updates its route table entry for the originating node by placing next hop towards destination and last hop node in the precursor lists (for respectively forward and reverse route).

RRepGratuitousGen: sends a gratuitous RRep to the Originator (if the gratuitous Flag is set in the RReq). It builds a route from destination to Originator, just as if the destination had issued a fictitious RReq for the originating node.

RRepProcess: takes action upon RRep arrival.

```
Create or updates a route to the previous hop.
Create or update forward route.
If (node is not originator) forward RReq
Else send message from queue for which node knows a route.
```

RRepFwd: forwards Route Reply if node is not the Originator (of the RReq), and a forward route has been created or updated. The node sends a new route reply to next node, which can be found by consulting the node's route table for the originator[1] of the RReq.

displayError: displays debugging message (with 3 levels of debug)

updateRreqBuff: scans for node's sent RReq messages and remove obsolete values.

scanTable: scans the node's routing table for a specified destination and returns its index, or "-1" if not found.

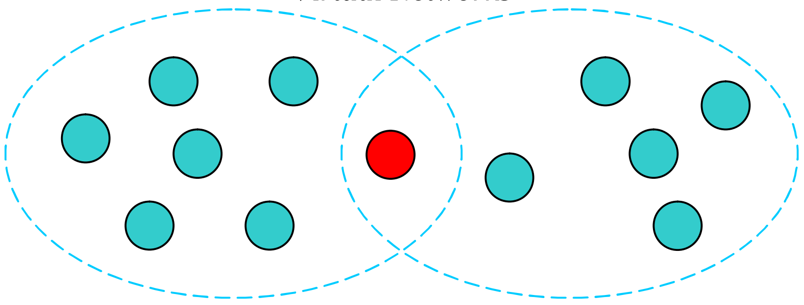
updateTable(entry): updates the routing table, by replacing the specified entry by a new one.

updateTable(): scans the routing table to remove timed out routes.

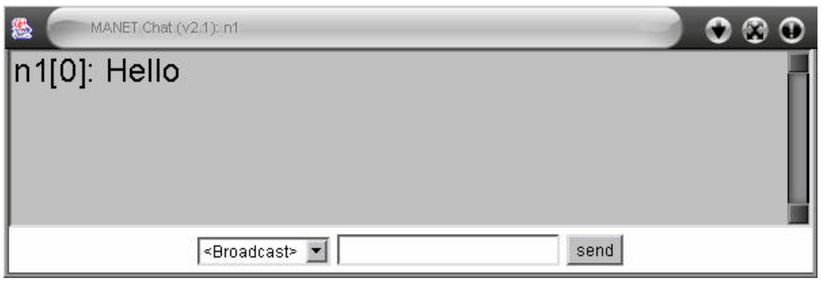
5.5. Testing

Included with the framework is a *chat*, and with the use of *virtual network*, it is possible to experience multihop routing. Virtual networks is a layer that allows to emulate networks with nodes that might not all see each other. For example, one can create two networks, with nodes belonging to one or the other (or both). The interest is that the node(s) belonging to both networks will have to act as forwarding nodes, for the others to communicate from a network to another. This is easier than to position nodes over a field to test multihop routing!

Virtual Networks



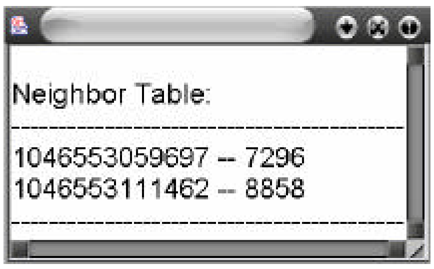
The chat is a simple application where people can send message to others by choosing a node's name from a list. It is necessary to do a broadcast at the beginning so that the lists are filled with node names (no possibility of entering a node's ID yet).
It shows 3 windows, one for the main chat, and one for each module (Neighbor, Statistics):



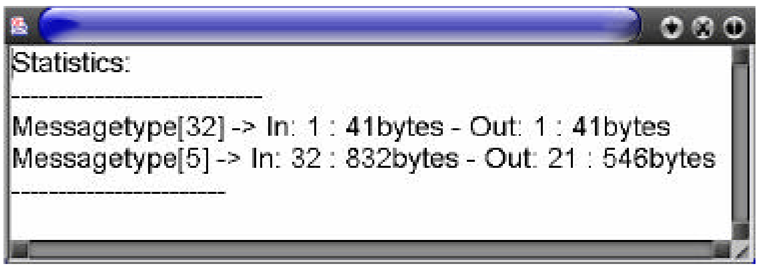
n1[0]: Hello

[1] The node for which the route request is supplied

Module windows:



```
Neighbor Table:
--------------------------------------
1046553059697 -- 7296
1046553111462 -- 8858
--------------------------------------
```



```
Statistics:
--------------------------
Messagetype[32] -> In: 1 : 41bytes - Out: 1 : 41bytes
Messagetype[5] -> In: 32 : 832bytes - Out: 21 : 546bytes
-----------------------
```

6. Still to be done

- *Generate and handle RReq's:* in order to detect the departure of a forwarding node, we have to implement RReq message generation. This can be done with using Reto Krumenacher's neighboring module which sends periodically Hello messages. This way, local repair can also be implemented.

- *Implement RReq retry*, using exponential backoff (to determine the waiting time for a RRep) and ring search technique (to determine TTL of RReq's) in order to not saturate the network. This would require a thread that would wait for a RRep, while the layer's thread remains dormant.

- *Add RRepAck feature* in configuration file (for unreliable or unidirectional links). This is to be certain RRep are not lost over the route discovery (as they are not resent).

7. References

- [1] Ad hoc On-Demand Distance Vector (AODV) Routing:
  Draft-ietf-manet-aodv-12.txt
  Charles E. Perkins, Elizabeth M. Belding-Royer, Samir R. Das
  http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-12.txt

- [2] DSR and AODV:
  Mobile Ad Hoc Networks Jim Thompson, Musenki
  http://nycwireless.net/presentation/jt_adhoc_tutorial.pdf

- [3] Mobile Ad Hoc Networks:
  http://lcawww.epfl.ch/Publications /Giordano/Giordano01a.pdf Sylvia Giordano

- [4] 802.11 and iPacks:
  General overview of IEEE 802.11 David Cavin
  http://lsrwww.epfl.ch/cavin/work/manet/presentation.pdf

8. Source code

In annex.