Discovering the network topology of a MANET

Stefan Thurnherr, 6th semester SSC

stefan.thurnherr@epfl.ch

Project Advisor

David Cavin

EPFL-LSR-IC

david.cavin@epfl.ch

June 2003

Abstract

Today Mobile Adhoc Networks are of more and more growing interest, on the one hand because they work without any infrastructure or control, on the other hand because the devices we use get smaller and smaller and thus the need to communicate to other devices can easily occur anywhere, at any time. But the need & desire to communicate immediately and always rises the question whether the interlocutor is reachable by any means, i.e. a Mobile Adhoc Network.

The LSR laboratory of EPF Lausanne (CH) developed a Java based framework for Mobile Adhoc Networks called FRANC which can be run on nearly any device from desktop computers to PDA's (apart some restrictions, depending on the JVM that is being used). This framework is based on a layer architecture and thus completely configurable with different functionalities.
This report is about an application layer written for this framework. Its aim is to discover the network topology of a Mobile Adhoc Network consisting of clients running the FRANC layer stack. The discovered topology may then be represented by a graph, thus making it possible to quickly get an overview of the reachable nodes - and to answer the question whether the desired interlocutor is reachable or not.

Table of Content

1 Introduction 4
1.1. About Mobile Adhoc Networks 4
1.2. About the MANET Framework 4
1.3. About this project 6
2 Gathering the topology information 7
2.1. Overview 7
2.2. The Viscovery Layer 7
2.2.1. How it integrates into the MANET Layer Stack 7
2.2.2. The configuration parameters 8
2.2.3. The message type TokenMessage 8
2.3. How to determine the next destination neighbor 9
2.3.1 LRV – Least Recently Visited 9
2.3.2 LFV – Least Frequently Visited 10
2.4. All about the TokenMessage 10
2.4.1. Inside the TokenMessage 10
2.4.2. A TokenMessage example using LFV 11
2.4.3. De-/Serialization of a TokenMessage 12
2.5. What Viscovery does to a Token 13
2.5.1. A perceived TokenMessage is not destined to this node 13
2.5.2 A perceived TokenMessage is destined to this node 14
3 Visualizing the topology information 15
3.1. Overview 15
3.2. The GUI 16
3.3. Evaluating different graph libraries 18
3.3.1. JGraph 18
3.3.2. GVF 18
3.3.3. And the winner is....TouchGraph 19
4 Futurework 21
5 Personal Conclusion 22
6 References 23

1 Introduction

1.1. About Mobile Adhoc Networks

Today Mobile Adhoc Networks are of greater and greater interest not only in the field of research, but also in terms of economy: They do not need any fixed infrastructure (i.e. no base antennas), they have the potential to be self-organizing and to increase their capacity with the number of users, and the denser such a net is, the lesser power we potentially need to transmit and/or receive, which is an interesting aspect in terms of battery life and device size.

1.2. About the MANET Framework

The MANET framework is (as of 2003) a project administered by David Cavin and Yoav Sasson under the direction of Professor Schiper of the distributed systems laboratory (LSR) at the Swiss Federal Institute of Technology in Lausanne (EPFL, http://www.epfl.ch).
The MANET framework has been entirely built through semester projects so far: the initial framework was programmed by Javier Bonny & Urs Hunkeler [1]. It consists of a layer architecture which is represented in its basic configuration in Figure 1.

An instance of this framework is configured by an XML file specifying which layers one wants to include in the layer stack. Each layer can be configured individually through some parameters; please refer to the corresponding documentation for details on the parameters of each layer (see References).

Some layers were outsourced to a whole semester project, namely:

- the AODV routing layer, done by Betrand Grandgeorge [2]
- the reliableLayer, done by A.Leiggener [3]: this layer is placed directly above the *VirtualNetworks* layer and requires an ACK to be sent by all neighbors of a one-hop broadcast.

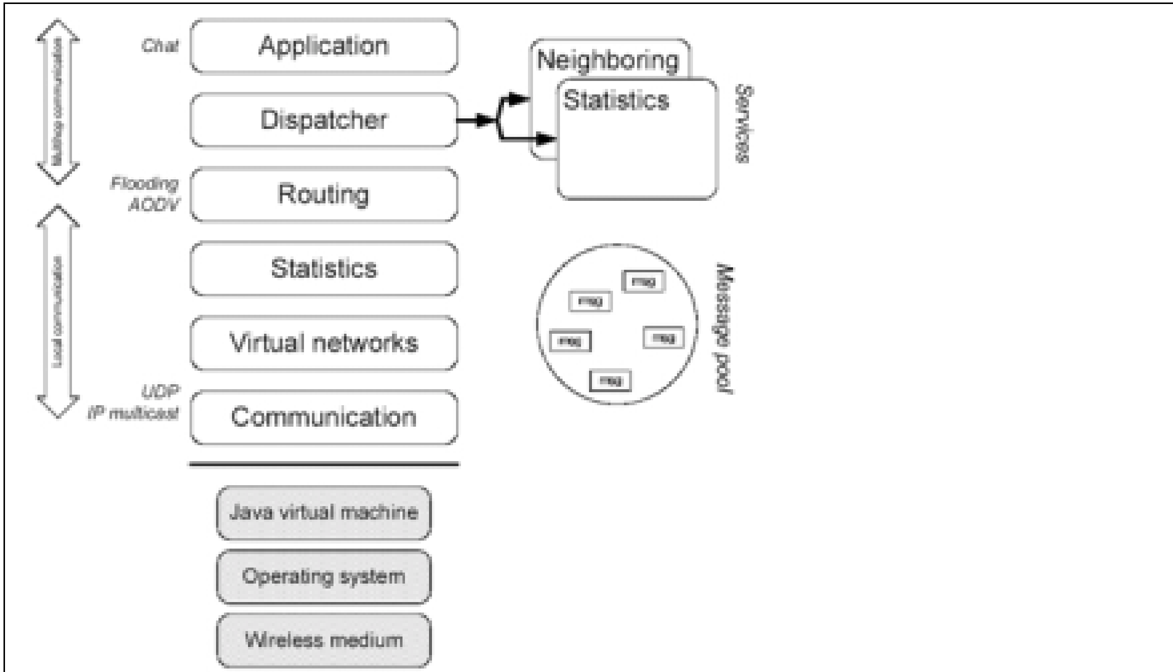


Fig.1: The MANET Framework in its basic configuration

Apart these layers the MANET Framework offers modules, called 'Services' in the above graphics. Currently two such modules exist and were designed during a semester project by Reto Krummenacher [4]: The *Neighboring module* continuously sends out HELO messages in order to discover the current neighborhood (all neighbor nodes reachable within the sending radius of a node) This information is vital for some layers as they need to know which nodes are reachable at a certain moment (routing, viscovery, chat). The *Statistics module* keeps track of the amount of data sent and received and is able to present the such collected data to any interested layer. Christophe Crausaz made extensive performance analysis in his semester project [5] using this Statistics module.

The third core part of the MANET Framework is the Message Pool. It is implemented as a thread of its own and used by all layers and modules whenever they need a new instance of an object registered within the framework via the XML configuration file: Instead of letting the Java Garbage Collector gather any such object that is not used anymore, the layers and modules put them into the Message Pool by calling its *freeMessage(msg)* method (see [1], *Part II, 2.1. MessagePool*). The Message Pool basically reinitializes them and makes them available to whichever layer asks for a new instance of a certain object. If a layer asks for an instance of an object that is not available yet, the Message Pool obtains one by calling the corresponding Constructor method. In this way the framework is able to recycle all used FRANC object instances.

1.3. About this project

The aim of this project was to build an application layer for the MANET Framework that is somehow able to gather information about the current network topology and to store the such retrieved information in order to be able to display some sort of visualization of the discovered network topology.

To gather such information this application layer (which I called *Viscovery*) creates and sends around a special message which we called *Token*, since it is thought to be unique within one connected network.
This *Token* stores information about each visited node so it can visualize the "current" network topology. Of course we can never get a graphical representation that reflects the current topology accurately since the *Token* holds the information that was valid for each node at the time of the last visit. But the target networks are not (yet) supposed to suffer neither from high mobility nor from partitions; such problems will be adressed to futurework.

The rest of this report consists basically of two parts, as the Viscovery application can be split up into the information-gathering algorithm on the one hand and the visual representation of the discovered topology on the other hand. After these two core parts, I will lose some words about futurework, then concluding this report by with some personal impressions & aspects and finally specify the references.

2 Gathering the topology information

2.1. Overview

In this chapter I will explain how the Viscovery layer integrates into the layer stack of the MANET Framework, what decisions have been made concerning the format of the message used by the Viscovery layer (the so-called *Token*), and finally what changes a node running the Viscovery layer makes to the content of a *Token*.

I will not cover any interaction that a user can undertake if the GUI is available; everything related to the graphical part of the Viscovery application is treated in Section 3, as the GUI can be turned off via the corresponding parameter in the XML configuration file without cutting off the basic, non-graphical functionalities described in this section.

2.2. The Viscovery Layer

2.2.1. How it integrates into the MANET Layer Stack

The Viscovery application consists of one single layer that is added onto the top of the MANET layer stack, just before the Absorbing Layer (see Figure 2).

The AsyncMulticast is the physical layer of the MANET Framework in order to be able to communicate with other nodes.
The Dispatcher is needed because we need a reference on the Neighboring Module, which can only be obtained by accessing the corresponding method of the Dispatcher.
The VirtualNetworks is needed because wireless transmission is equivalent to Multicast transmission, i.e. a package that we want to transmit is being sent out into the air and can be perceived by all neighbor nodes. The MANET Framework does not rely on any routing or filtering done outside the layer stack, so every perceived package will get into the layer stack. But if we want to simulate a multihop network within a space-limited location (typically smaller than the sending range of the nodes), we need a layer that filters the packages that normally would not be perceived by certain nodes; this is exactly what the VirtualNetworks Layer does.
The Viscovery Layer is then set above this existing stack, and finally we set up the Absorbing Layer which simply consumes all messages that reach the top of the stack, in order to prevent a buffer overflow due to unknown message types.

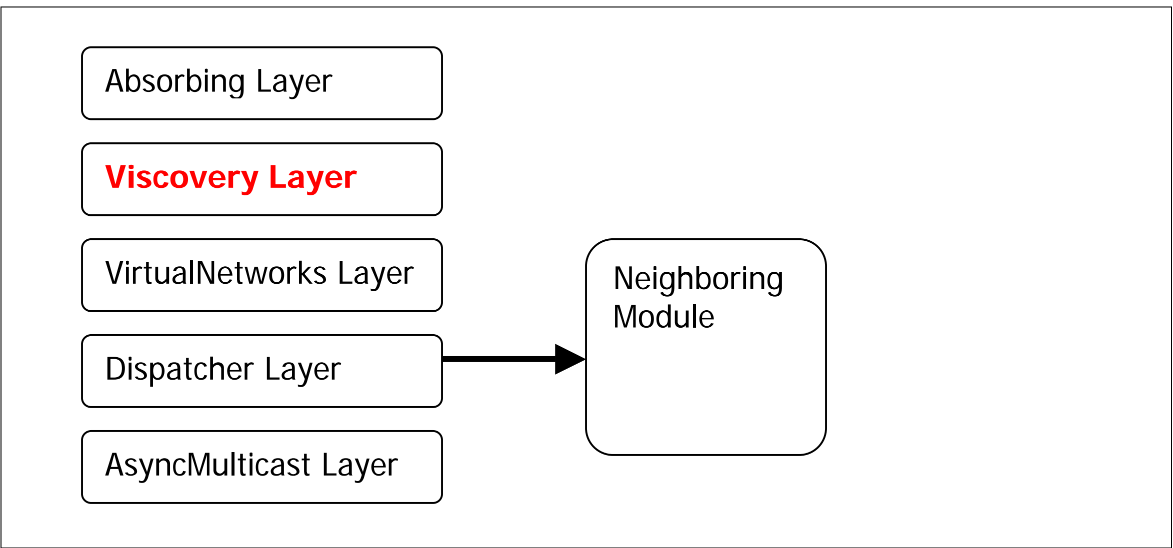


Fig.2: Minimal MANET configuration to successfully run the Viscovery Layer

2.2.2. The configuration parameters

The parameters for the *Viscovery* Layer that can be specified in the XML configuration files are the following:

Parameter name	*Possible value(s)*	*Effect*
TTL	Any positive integer	The maximal number of hops that this message should survive; Because each TokenMessage gets newly initialized at the time of deserialization, this number has no influence or effect, but could potentially be used in a future version that relies on multihop.
MsgType	*Token*	The message type associated to this layer. This parameter has to be set to *Token*, or the Viscovery Layer will not work correctly.
FrameVisible	true / false	This parameter decides whether the GUI is visible and active or not. If it is set to *false*, then no interaction is possible, i.e. you cannot create or hold a *Token*, you cannot change the baseDelay, and you cannot see the graph of the discovered network topology. But nevertheless the basic functionality (i.e. to receive, handle correctly and send a *Token*) of the Viscovery Layer is not affected.
baseDelay	Any positive integer (recommended between 1000 and 10000 [msecs])	The delay that should be applied when the treatment of a *Token* is finished, but before the *Token* is sent downward the FRANC Layer stack. The number is interpreted as milliseconds, and is especially useful while debugging or following a *Token's* trace.

2.2.3. The message type TokenMessage

The Viscovery layer has its own type of messages, the *TokenMessage*. This class inherits from the superclass *Message* as for all message types within the MANET

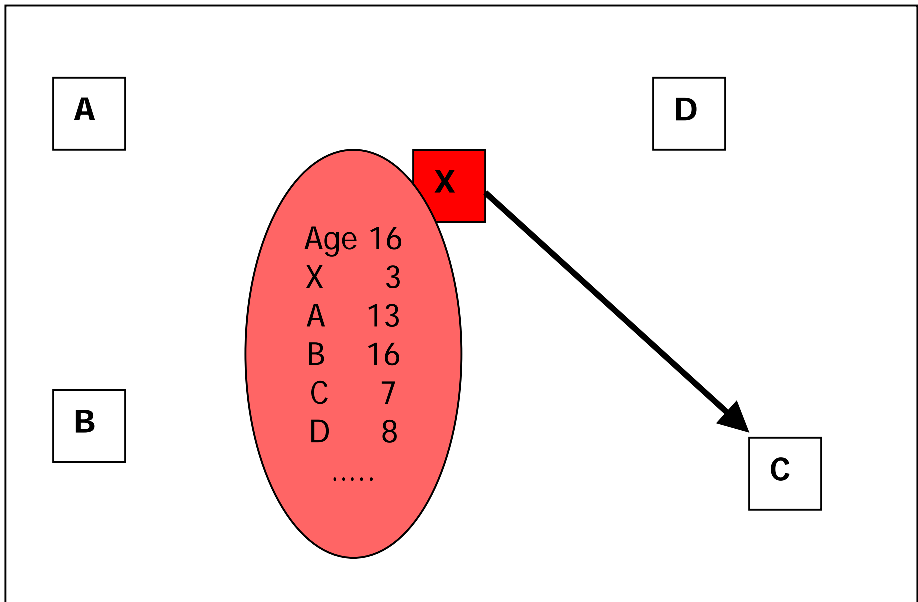
Framework. This causes a message of type *TokenMessage* to be propagated up the stack until it reaches the Viscovery Layer (given of course that the message is destined to this node). The Viscovery Layer recognizes the message to be a *TokenMessage* and therefore does not hand it further up the stack; instead the *handleMessage()* method (see [1], *Part II, 2.2.4. handleMessage()*) of the Viscovery Layer picks it out and the treating of the *Token* is started by calling the Viscovery's *treatMessage()* method (see *2.5. What Viscovery does to a Token*).

2.3. How to determine the next destination neighbor

Before sending a *TokenMessage* towards its next destination, we first need to determine this next destination. To achieve this we decided to implement two different algorithms (*LRV* and *LFV* - see the corresponding subsections for details) which are both local algorithms, i.e. the next destination is always one of the direct neighbors of the node that is currently being visited. The neighborhood information is obtained by the Viscovery Layer from the Neighboring Module [4].

The decision which algorithm will be used must be made at creation time of a *Token*, since this influences the meaning of the counter variables associated to each node. A *Token* can only be created by a node running the GUI-enabled version of the Viscovery Layer, i.e. the parameter *frameVisible* in the XML configuration file needs to be set to *true*).

2.3.1 LRV – Least Recently Visited



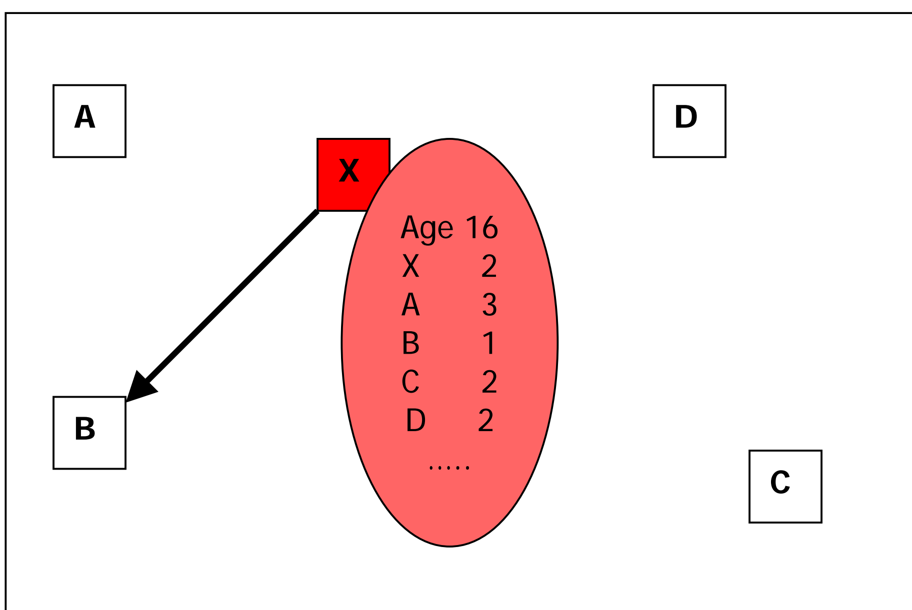
The *Token* (configured to use LRV and represented by the semi-transparent ellipse) currently visits node X and has visited 16 nodes so far. In order to determine the next neighbor, LRV runs through the *Token*'s tokenTrace and searches for the smallest value (i.e. the node that has not been visited for the longest time). In this case this would be node C. Then the *Token*'s age is increased by 1 and the counter associated to node X is set to the current age of the *Token*. Finally the *Token* is sent to node C.

Fig.3: An example how the next destination node is chosen according to LRV.

This algorithm selects the next destination out of the possible nodes (i.e. nodes that are within the sending range of the currently visited node) that have not been visited for a longer time than any other possible node. A small example is depicted in Figure 3.

2.3.2 LFV – Least Frequently Visited

This algorithm selects the next destination out of the possible nodes (i.e. nodes that are within the sending range of the currently visited node) that have been visited fewer times than any other possible node. In Figure 4 I show an example to clarify this algorithm.



The *Token* (configured to use LFV and represented by the semi-transparent ellipse) currently visits node X and has visited 16 nodes so far. In order to determine the next neighbor, LFV runs through the *Token*'s tokenTrace and again searches for the smallest value (i.e. the node that has been visited the fewest times). In this case this would be node B. Then the counter associated to node X as well as the age of the *Token* are increased by 1 and finally the *Token* is sent to node B.

Fig.4: An example of how the next destination node is chosen according to LFV.

2.4. All about the TokenMessage

As mentioned earlier in this document, its own message type is associated to the Viscovery Layer, called *TokenMessage* (inside the MANET Framework this message corresponds to the message type #33). In this subsection I talk about the structure of a *TokenMessage*, about its De-/Serialization and finally I will present an example featuring the path of a *TokenMessage* visiting 5 nodes.

2.4.1. Inside the TokenMessage

Beneath the header, which is commun to all messages inheriting from the superclass *Message* (see [1], *Part II, 2.2. Message*), the *TokenMessage* contains the following data:

String	tokenAlgo	Stores the *Token*'s algorithm, i.e. *LRV for Least Recently Visited* or *LFV* for *Least Frequently Visited* - see *2.3. How to determine the next destination neighbor*
List	tokenTrace	The history and core part of a *TokenMessage* : For each node that was visited by this *Token* at least once, the Viscovery Layer of the visited node adds a row to the tokenTrace containing the nodeID of this very node, its counter value (whose meaning depends on the configured tokenAlgo) as well as all the nodeID's of the direct neighbors of that node at the time of visit. The counter and the neigbhor's nodeIDs of a visited node are then kept up to date by verifying them at each visit. For the exact sequence, see *2.5. What Viscovery does to a Token* .
Int	currentNodeIndex	This field contains the tokenTrace index of the row that is associated to the currently visited node; it is therefore set to a valid value only once a *TokenMessage* arrives at its destination node, and is not serialized when transmitting a *TokenMessage*.
long	nbVisits	Represents the global counter or age for this *TokenMessage*. This number is equal to the number of nodes that this *Token* has visited so far. It is incremented just before a *Token* leaves a node towards its next destination node; during a visit, the current visit is not counted in yet.

2.4.2. A TokenMessage example using LFV

In the following I'll give an example of a topology and the corresponding tokenTrace (configured to LFV) after a certain sequence of visits. For reasons of simplicity we assume that the nodes of the considered network do not move out of their mutual sending range and therefore the topology graph stays static. So let us start from the configuration as depicted in Figure 5.

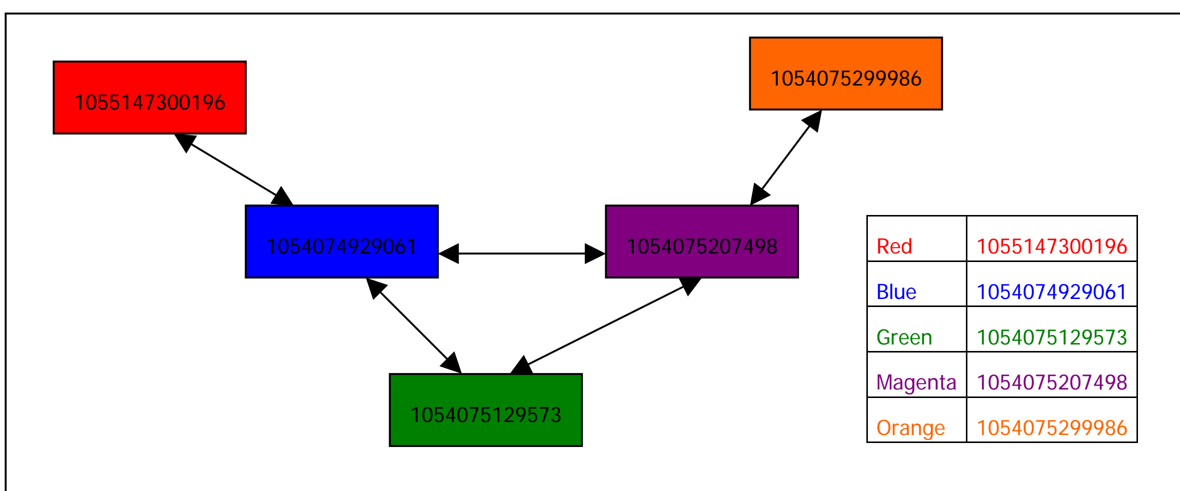


Fig.5: The network configuration of the discussed example of *Token* passing using LFV.

Now assume that a *Token* is generated at the red node and goes along the path shown in Figure 6, which is a perfectly correct path according to LFV:

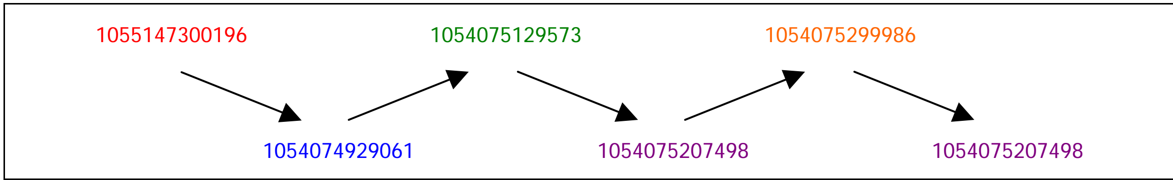


Fig.6: An imaginative, but correct path of the *Token*, according to LFV.

Once the *Token* arrives at the magenta node for the second time, we get the tokenTrace as sketched out in Figure 7. Some minor differences may occur regarding the order of the neighbors for each node because this is somewhat arbitrary (it actually depends on the neighbortable supplied as a static array by the neighboring module [4]). But the same goes for the choosing of the next destination at the blue node when the green and the magenta node have the same count values, i.e. they were not visited yet by this *Token*.

nodeID	*counter*	*Neighbor 1*	*Neighbor 2*	*Neighbor 3*	*Neighbor 4*	*Neighbor 5*
1055147300196	1	1054074929061				
1054074929061	1	1055147300196	1054075129573	1054075207498		
1054075129573	1	1054074929061	1054075207498			
1054075207498	2	1054075129573	1054074929061	1054075299986		
1054075299986	1	1054075207498				

tokenTrace

Fig.7: The *Token* data, after the traversal of a path as depicted in Figure 6.

The next destination node would again be determined according to LFV. It would be either the green or the blue node.

2.4.3. De-/Serialization of a TokenMessage

The De-/Serialization is done in the following order, starting from the top:

String	tokenAlgo	(for explanations on this *TokenMessage* field, see 2.4.1)
long	nbVisits	Represents the global counter or age for this *Token*. This number is equal to the number of nodes that this *Token* has visited so far. It is only incremented just before the *Token* leaves a node; during a visit, the current visit is not counted in yet.

int	nbTraceRows	The number of rows (or lines) of the tokenTrace. For each node that was visited by this Token at least once, the Viscovery Layer adds a row to the tokenTrace containing the nodeID of the corresponding node, its counter value (with respect to the tokenAlgo) as well as all the nodeIDs of the direct neighbors of that node at the time of visit.
int	rowCapacity of row #0	The number of columns contained in the tokenTrace's first row (index 0).
long	from row[0] to row[rowCapacity – 1]	The first number is the nodeID of the node which this row refers to; the second value is the counter associated to this node; the rest of the row consists of the nodeIDs of the neighbors of this node.
int	rowCapacity of row #1	The number of columns contained in the tokenTrace's second row (index 1).
long	from row[0] to row[rowCapacity – 1]	The first number is the nodeID of the node which this row refers to; the second value is the counter associated to this node; the rest of the row consists of the nodeIDs of the neighbors of this node.
:	:	:
:	:	:
int	rowCapacity of the row #(nbTraceRows-1)	The number of columns contained in the tokenTrace's last row (corresponds to index (nbTraceRows-1)).
long	from row[0] to row[rowCapacity – 1]	The first number is the nodeID of the node which this row refers to; the second value is the counter associated to this node; the rest of the row consists of the nodeIDs of the neighbors of this node.

2.5. What Viscovery does to a Token

We need to distinguish 2 cases, namely whether a perceived *TokenMessage* is destined to this node or not.

2.5.1. A perceived TokenMessage is not destined to this node

The Viscovery layer is prepared to treat this case because we cannot rely on the existence of a routing/filtering algorithm since layers in the MANET Framework can easily be deactivated through the XML configuration file. As a consequence the Viscovery Layer potentially receives all *TokenMessages* sent out by nodes within this node's sending range. And since we can profit from the more up-to-date content of these *TokenMessages* to keep our graph as most up-to-date as possible, it would be silly to simply discard these messages. So if (and only if) the GUI is enabled, the Viscovery layer evaluates the content of any *TokenMessage* not destined to this node and updates the graph accordingly. Once this is done the message is dropped immediately (i.e. passed over to the Message Pool [1]); the content of such a *TokenMessage* is in no way changed, neither is the message reintroduced into the layer stack.

2.5.2 A perceived TokenMessage is destined to this node

If the destination of the message equals the nodeID of this node, a *TokenMessage* undergoes the following treatment while staying at the Viscovery layer of this node :

msg.visitNode(thisNodeID)	First of all this method checks whether this node has already an associated row in the tokenTrace, i.e. has already been visited at least once by this *Token*. If no such row exists, a new row is appended to the tokenTrace with this node's nodeID and zero as the counter's value. Next, this method investigates the algorithm of the *Token* (see *2.3. How to determine the next destination neighbor*) and reacts accordingly: LFV: The counter associated to this node's nodeID is increased by 1, as it represents the number of times that this node has been visited. LRV: The counter is set to the value of the global age/counter of the *Token* and then increased by 1, as the global counter is only increased at the time of departure from this node. The counter value associated to this node's nodeID represents now the time of the most recent visit (i.e. the current visit).
msg.updateNeighbors(currentNeighborTable)	This method is called to update the neighbor entries associated to the nodeID of the node that is currently being visited. On the one hand, for every entry (nodeID) in the currentNeighborTable (which is obtained from the Neighboring Module [4]), Viscovery checks whether that nodeID is already present in the row of the tokenTrace that is associated to this node. On the other hand every entry in the tokenTrace's row of this node is tested for validity, i.e. if the node corresponding to that nodeID is still a direct neighbor of this node. At the end the row of the tokenTrace associated to this node contains the same nodeIDs as the currentNeighborTable.
msg.updateOthersNeighbors()	We do not only want to keep up to date the information whether any node is a neighbor of this node, but also whether this node is a neighbor to any other node; this is exactly what this method does: It runs through all rows of the tokenTrace except the row that is associated to this node, verifies any occurance of this node's nodeID on the one hand, and adds this node's nodeID wherever necessary on the other hand.
visFrame.updateGraph()	This method is only called if the configuration parameter *frameVisible* is set to *true*. In this case the graph that is currently being displayed within the GUI is cleared and a new graph is created with the information of this *Token*.
getNextDestination()	This method obtains the current Neighbor Table from the Neighboring Module [4] and chooses one of its nodeIDs to be the next destination node. The selection is done with respect to the algorithm of the *Token*, the current counter values of all nodeIDs figuring simultaneously in the tokenTrace and the currentNeighborTable as well as any current neighbor not figuring yet in the tokenTrace.
send(nextDestination)	Finally the Viscovery Layer sends the treated *TokenMessage* downward the FRANC Layer stack back to the network via this method, which itself calls the method *sendMessage(msg)* inherited from the superclass Message...and there the *TokenMessage* goes!

3 Visualizing the topology information

3.1. Overview

This chapter deals with the other core part of my semester project, which is the Graphical User Interface (GUI) and the displaying of the discovered network topology.

Keep in mind that this GUI is optional; the user can decide whether he wants to have all the graphical features described in this chapter by simply setting the corresponding parameter in the configuration file. However the basic functionality (as described in the preceding chapter) of the Viscovery Layer depends in no way on the GUI.

Fig.8: Screenshot of the Viscovery GUI with the log window, just after startup

3.2. The GUI

When starting up the application, the GUI looks as depicted in Figure 8.

We can change the tab located at the top of the GUI to show the current graph instead of the log window (see Figure 9 for the corresponding screenshot). No *Token* has been received or generated yet, so the graph shows only the node on which this Viscovery Layer is running.

Fig.9: Screenshot of the Viscovery GUI with the graph window, just after startup.

Now let us have a closer look at the buttons and drop-down lists in the bottom:

set sending delay [s] to: 3 (default) 3 (default) 1 3 5 10	This first line of the buttons area allows to set the delay (measured in seconds) that is applied once the next destination neighbor is evaluated, but before the *Token* is actually sent to this next destination neighbor. Set this delay to a higher value if you want to slow down the *Token* circulation. Keep in mind that this delay is set individually for each node (*see also 2.2.2. – The configuration parameters*).
Choose algo Choose algo LRV LFV	The next drop-down list must be used in order to be able to generate a new *Token*; it sets the algorithm according to which the *Token* will choose its next neighbors. The choices are: • LRV - Least Recently Visited (see subsection *2.3.1. LRV*) • LFV - Least Frequently Visited (see subsection *2.3.2. LFV*)
LFV Create a new Token	Once you have chosen the *Token's* algorithm, the button *Create new Token* will get enabled and you can generate a new *Token* by clicking on this button. In the log window you will then see the header and the data of this newly created *Token* as well as the next destination node, indicating that the *Token* left this node towards that destination node. You can change to the graph tab to see already a 2-nodes-graph, i.e. this node and the next destination node of the *Token* you've just created and sent (if the Neighboring Module already detected more neighbors, they too will show up in the graph). If no neighbor is available, the log file indicates this and the *Token* is destroyed.
Clear log window	This button is quite self-explaining; Even if the log window has an autoscroll function, it is sometimes useful to clear the log file and to restart the logging.
Hold next visiting token	The third line of buttons can be used to hold back a *Token* before its new destination node is evaluated. This checkbox may be activated at any time and will prevent the next *Token* arriving on this node and trying to evaluate the next destination node from doing so and being sent towards that node. If the checkbox is active, the two buttons next to it will get enabled:
Release the token again Destroy this token	As long as no *Token* is actually being hold back, it is possible to disable the holding back of the next *Token* by simply clicking either the checkbox again or one of these two buttons. This will set the *hold-back flag* to *false* and these two buttons will be disabled again. Once a *Token* was received (or generated) on this node, it is consequently hold back, i.e. the header is displayed in the log window and the old graph is updated with the new data contained within this *Token*. But the determination of the next neighbor does not happen until the *Token* is released by either deactivating the checkbox or simply clicking the correspondingly labeled button. The button *destroy this token* is put at one's disposal if one wants to eliminate a *Token* that is currently being hold back by this node. When clicking this button the Viscovery Layer immediately drops this *Token* without treating it furtherly.

These possibilities of interaction are independent of the tab that is currently selected, i.e. you can change the selected tab at any time without changing the state of any button, checkbox or drop-down list.

3.3. Evaluating different graph libraries

As I spent quite a lot of time on choosing an appropriate graph library to display the discovered network topology, I want to loose some words about this topic.

3.3.1. JGraph

The first graph library that I seriously considered as being suitable for the aim of this project was *jgraph* [6]. The project was started by Gaudenz Alder, first as a Semester Work, then continued as a Diploma Thesis at ETHZ (Swiss Federal Institue of Technology Zurich), before finally being released as an open source projet during 2002.

The intention of this project is to provide a freely available, standards-compliant and thoroughly documented open source component to display and edit graphs (networks) with Java.

[quotation from the jgraph website]

The main advantages compared to other graph libraries (even with the one that I finally used) are that it is guaranteed to be 100% Java Swing compatible, it is designed in a clearly and efficiently way (according to its website), and it is very well documented.

There are nevertheless a few important drawbacks which caused me to abandon this library; the main disadvantage was that no layout algorithm was included in the jgraph distribution (as of May 2003), and to develop such an algorithm by myself would have gone beyond the scope of my semester project.

3.3.2. GVF

A second graph library that showed promise is an open-source project called GVF: The Graph Visualization Framework [7].

The Graph Visualization Framework is a set of Java 2 packages that can serve as a foundation for applications that either manipulate graph structures or visualize them. The libraries implement several basic modules for input, graph management, property management, layout, and rendering.

[Quotation from the GVF website]

It seems that GVF would perfectly suit our needs. The problem with this library is that it is very poorly documented and is intended to be rather used as-it-is, i.e.

with the provided GUI and functions (which enable the user to create and edit graphs). I adressed my need to use GVF as a graph library in the forum provided for this sourceforge [8] project, and I always received answers on my questions within one or two days. But even after 2 weeks of trying to get to run a "hello-world"-like graph I did not succeed. Finally I had to abandon the idea of using GVF as the graph library for Viscovery despite the enthusiasm I had (and stll have; see futurework) about it fitting perfectly my needs.

3.3.3. And the winner is....TouchGraph

The graph library I finally used for the current version (v1.0) of the Viscovery Layer is the one from the TouchGraph project [9]:

TouchGraph is a set of interfaces for Graph Visualization using spring-layout and focus+context techniques.

[quotation from the TouchGraph website]

TouchGraph has its name from the layout algorithm it uses to arrange automatically all nodes in a suitable manner. In fact the nodes execute a force on each other which pushes them off mutually. In this way we get a neat view of a graph, at least as long as a physical model of the corresponding graph would stay in two dimensions: Imagine that we replace each node by, let's say, some big visible electron and put the whole model in a place with no gravitation. On the one hand the electrons would repel each other, and on the other hand the connections between the nodes/electrons would hold the graph together. As a consequence we would get after a while a stable model, which will be either two- or three-dimensional, depending on the number and location of the connections between nodes/electrons; if it is three-dimensional and we project it onto 2 dimensions (which is done by displaying the graph on the screen), the resulting picture will not always be comprehensive and demonstrative because a lot of overlapping will occur.

As a consequence the TouchGraph layout algorithm works very well for networks that are poorly populated and distributed over a rather large geographical area, i.e. only 2-3 nodes are within the sending range of each node. But the great advantage of this layout algorithm can easily become a disadvantage when lots of nodes are within the same sending range because in this case the described algorithm will produce a lot of overlapping and the expressiveness will decrease rapidly.

It is also for the reasons explained above that the node labels consist of only the last four digits of the corresponding nodeID. The longer the node labels are, the more probably they overlap and thus causing the graph to become illegible. In Figure 10 are depicted examples of two network configurations, each with 11 nodes; you'll easily recognize the problem of the TouchGraph algorithm as discussed above.

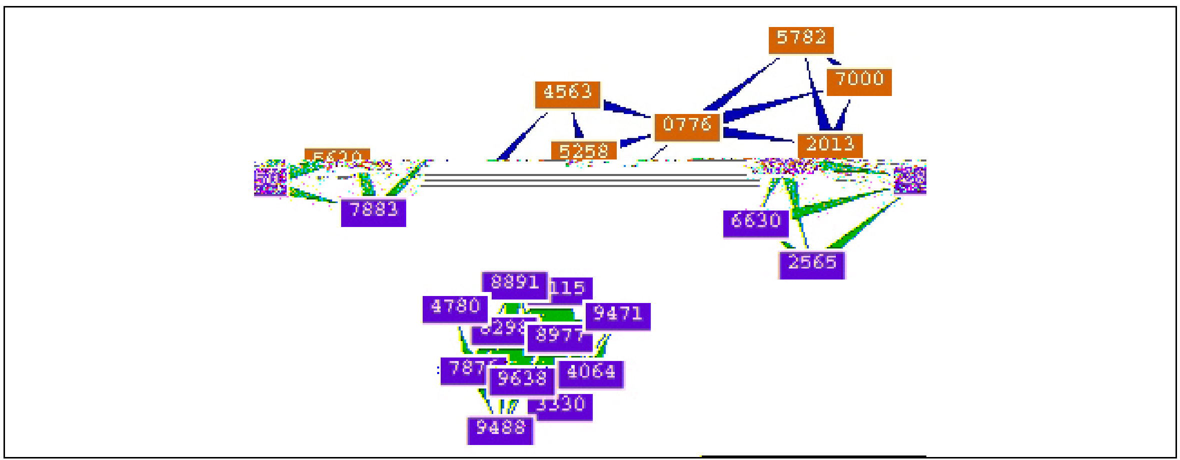


Fig.10: Top: The discovered network topology of nodes that are spread out over an area clearly larger than the sending range of each node.

Bottom: The discovered network topology of 11 nodes which are situated very closely, i.e. every node can reach every other node within one hop.

4 Futurework

The main future work will certainly consist in extending this application to be more interactive. Based on the graph, I could imagine for example a chat application: Assume user A* sitting behind the computer labeled node A. By clicking on a node B of the node A's graph, a window pops up which allows the user A* to type a message that is sent to the user B* sitting behind the computer constituting node B. On the graph of user B*, the message arriving from user A* is somehow indicated within the icon representing the node A.

Another application within the graph could be to show statistics about the throughput or the signal strength toward every other node. When a user A* on node A pauses his mouse over a certain node C of the displayed graph, a popup appears indicating how many packets have been sent to this node C and how good the connection is (in terms of signal strength, of packet loss etc...).

A third promising extension would be to implement a routing algorithm based on the discovered network topology. So if node A wants to communicate to node D, the nodes we have to visit to find the shortest way from node A to node D are looked up in the topology graph existing on node A. Of course this involves a number of problems (delay, partitions, etc.), but the idea seems promising.

But besides these extensions to the existing Viscovery application, there are a few aspects that could be improved within the existing application. Namely I am still convinced that GVF as the graph library would be the better choice; to clear this, it should first be possible to compare the performance of the TouchGraph vs. the performance of a GVF implementation, with regard to the needs of the Viscovery application. To get a detailed insight on the problems encountered when I tried to implement GVF, I suggest you start with the forum on [7]. The thread I started within the *help forum* is called *how to create my own graph*, the direct link is http://sourceforge.net/forum/forum.php?thread_id=873265&forum_id=126558 (as of June 2003).

Another aspect that I did not really test is how the application behaves on the occurance of partitions or multiple *Tokens* within the same connected network. As long as the application is only used for small manageble networks, these aspects are of minor interest. But once we want to simulate real networks, such cases must be taken into consideration.

5 Personal Conclusion

The field of Mobile Adhoc Network is an emerging field of studies with a bright future. Thus it was a great possibility for me to gain an insight into this subject, and I am sure it will not be the last time I worked on this, as I am still, or even more interested in these networks.

Apart my enthusiasm about the subject, it was a very valuable experience to have to design an application from the scratch, i.e. starting with some theoretical lecture about the subject, going over to think about and plan the implementation, then to implement the designed patterns and algorithms, and finally creating a GUI to enable users to steer the application.

I had to learn that once the application is finished, there are still tests to do which can take up a lot of time and thus should not be underestimated. Further I had to deal with the problem that the graph library that I intended to use in the beginning turned out to be suboptimal, and needed to be replaced by a better one (TouchGraph). And on the contrary in the final weeks of the project, I had to stop being enthusiastic about implementing the GVF library, even if success was close; the time was just not sufficient to continue with the GVF attempts.

To conclude, I would like to thank my project advisor David Cavin. On the one hand he gave me plenty of rope and he did never impose his will on me, and on the other hand he had always time to help out when I had a problem without losing his patience. It is for sure that this contributed much to the success of this project.

6 References

[1] J.Bonny & U.Hunkeler, "MANET Framework", February 2003

[2] B.Grandgeorge, "AODV routing algorithm for multihop Ad Hoc Networks", February 2003

[3] A.Leiggener, "Reliable Broadcast at 1 Hop in FRANC", June 2003

[4] R.Krummenacher, "Use of Modules for better inter-layer communication", February 2003

[5] C.Crausaz, "Design and implementation of a performance evaluation mechanism for FRANC", June 2003

[6] The JGraph Project, http://jgraph.sourceforge.net/

[7] GVF – The Graph Visualization Framework, http://gvf.sourceforge.net/

[8] Sourceforge is the world's largest Open Source software development website, http://www.sourceforge.net/

[9] TouchGraph – http://touchgraph.sourceforge.net/