Use of Modules for better inter-layer communication

Reto Krummenacher
SSC5
reto.krummenacher@epfl.ch

Project Advisor
David Cavin Yoav Sasson
david.cavin@epfl.ch yoav.sasson@epfl.ch
EPFL-IC-LSR

February 2003

Abstract

Ad-hoc networks are a key in the evolution of mobile networks. They are typically composed of equal nodes communicating without central control. This dynamic topology brings new problems, like configuration advertisement , discovery and maintenance, as well as addressing and self-routing. While in the past telecommunication networks were studied and developed in separate building blocks, for mobile ad-hoc networks the interaction between higher and lower layers is essential to the user.
This report shows the use and the development of modules based on the LSR-MANET communication framework developed at EPF Lausanne. Modules are communication stack independent elements of the framework which are especially practical to collect and process data which shall be accessible to all layers. This data can consist of topology or power control information, network load or other statistical facts which help to optimize the network.

Contents

1 Introduction

The working environment for this project was defined by several limiting factors. The major limitations were given by the mobile ad-hoc network which allows only limited knowledge about the network topology and has significantly lower capacity than their wired counterparts. Moreover an implementation was desired for the iPaq Pocket PCs of the LSR mobile network.

1.1 Ad-Hoc Network[1]

Ad-hoc wireless networks (**M**obile **A**d-hoc **NET**work) inherit the traditional problems of wireless and mobile communications, such as bandwidth optimisation, power control and transmission quality enhancement. In addition, the multi-hop nature and the lack of fixed infrastructure, as well as the constantly moving topology, generate new research problems such as configuration advertising, discovery and maintenance, as well as ad-hoc addressing and self-routing.
In mobile ad-hoc network, topology is highly dynamic and random - topology changes may thus be hard to predict. Further characteristics are:

- Mobile ad-hoc networks are base on wireless links, which have a significantly lower capacity than their wired counterparts
- Mobile ad-hoc networks are affected by higher loss rates
- Mobile ad-hoc networks nodes rely on batteries or other exhaustible means for their energy. As a consequence, energy savings are an important system design criterion. The set of functions offered by a node is thus depending on its available power.

In addition to the problems encountered with infrastructured mobile networks (WLAN, GSM), nodes in mobile ad-hoc networks are confronted with a number of problems, which are otherwise solved by the base station:

- Power management is of paramount importance
- Information as node distribution, network density, link failure, etc..., must be shared among layers, and the MAC layer and the network layer need to collaborate in order to have a better view of the network topology and to optimize the number of messages in the network.

While in the past telecommunication networks were studied and developed in separate building blocks, for mobile ad-hoc networks the interaction between the higher layers and lower layers is essential to the users.

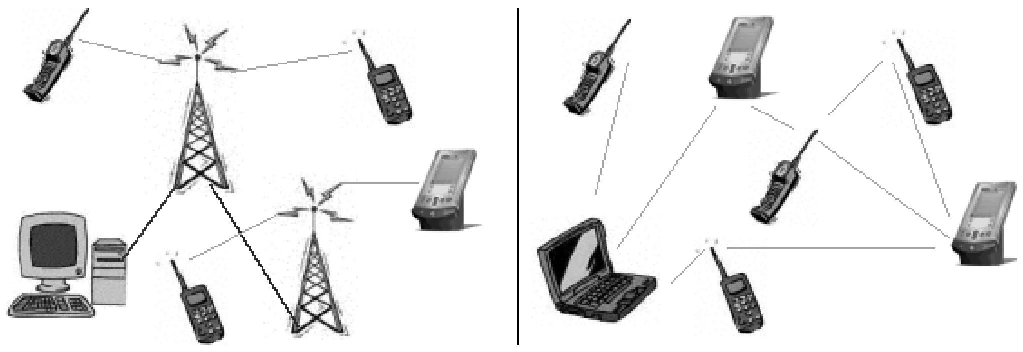

Figure 1: Mobile Network (WLAN,etc...) vs. MANET

1.2 iPaq 3660 - Pocket PC with Compaq WL110 Wireless PC card

The working environment is not only limited by the ad-hoc character of the network but also by the hardware provided. All programs are written in Java and run by means of a JVM called CrEme (produced by NSIcom, Israel). CrEme was created and optimized for the use with Pocket PCs. There is nevertheless a limiting factor coming with the virtual machine; it is based on jdk 1.1.8 and not on the latest version of Java. This is for example the reason why the class Vector and not another collection, e.g. ArrayList is used for the implementations of this project [see section 3].
The JVM is installed on the iPaq 3660 from Compaq:

- Operating System: Windows Powered PocketPC, Linux
- Processor: 206 MHz Intel StrongARM 32-bit RISC
- Memory: 32 MB RAM, 16 MB ROM
- Development environment: Visual C++, Java

From the above named facts it can be seen that the iPaq has other major limitations compared to contemporary computers: limited CPU and memory for example. This corresponds actually very well to the mobile ad-hoc characteristics named in section 1.1. Short, the iPaq is an easy to use and generic device able to concretely build wireless networks.[2]

2 The LSR-MANET Communication Framework

The communication framework is based on a stack of asynchronous layers [Figure 2]. The lowest layer is the communication layer, while the top-layer is the application. In between the user has free hand to add and remove layers. The composition of the stack is defined in the configuration file. In addition to the asynchronous layers, it is possible to add modules. Modules are independent of the communication stack and allow data access for all layers independent of its position. For more information the reader is referred to [3].

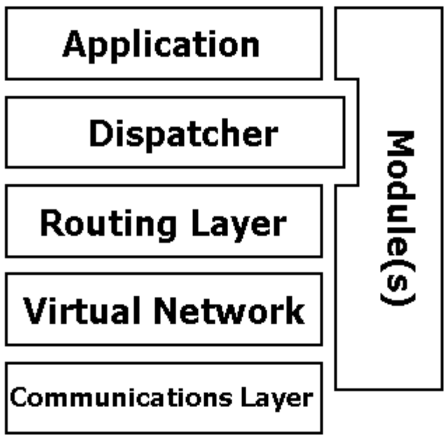

Figure 2: LSR ad-hoc communication framework (communication stack)

2.1 Asynchronous Layers

The LSR-MANET communication stack is based on a number of independent layers. Independent in the sense that they can be put together as pieces of LEGO, in a non-static order. The only layer that has its fixed place is the communications layer which is the interface between the stack and the communication media (Ethernet, air, etc...). Every class that wants to be an asynchronous layer has to extend the abstract class AsynchonousLayer[1]. This abstract class demands the implementation of two methods

- public void initialize(IConfiguration conf, CommunicationsLayer com)
- public void startup()

They are used to prepare and start the layer's activities. As the names already tell, initialize() is called upon bootstrap of the system, startup() upon bootstrap of the layer. For startup() it is in general enough to start the thread[2]. Initialize() on the other hand serves to do all the preparing necessary:

- initalization of local variables.
- definition of local parameters.
- lecture of configuration information from the configuration file (manet.config) [4] by use of the IConfiguration interface provided to the method.
- local storing of elements of the communication stack framework, as dispatcher, message pool, communication layer, etc...

Further methods implemented by every asynchronous layer are:

- public int sendMessage(Message msg)
- protected void handleMessage(Message msg)

These two methods are used for the transport of message inside the communication stack. SendMessage() passes the messages on to the lower layer in a synchronous manner, while handleMessage() is called by the lower layer to put the message in the buffer of this layer. In that way the messages are travelling in a asynchronous manner when climbing the communication stack.
The construction of the communication stack is done at bootstrap. The order and number of layers is defined in the configuration file. During the bootstrap phase every layer gets linked to its lower layer to which it can pass outgoing message by use of sendMessage(). A typical implementation of sendMessage() and handleMessage() is given in algorithm 1. This is the code used in the class AsynchronousLayer which in general does not need to be overridden[3].

Algorithm 1 sendMessage() and handleMessage() as implemented by AsynchronousLayer

```
protected void handleMessage(Message msg) {
    buffer.add(msg);
}

public int sendMessage(Message msg) throws SendMessageFailedException {
    return lowerLayer.sendMessage(msg);
}
```

[1]This is not the case for the CommunicationsLayer which is thus not an asynchronous layer.
[2]The AsynchronousLayer class extends Thread and is therefor itself a thread that has to be started.
[3]The methods are not defined abstract and thus automatically inherited.

2.2 Modules

Modules are communication stack independent elements of the framework. In that way their information is accessible for all layers belonging to the framework at all time. This allows a better and faster information flow between the different elements of the communication stack. As mentioned in the introduction, this interaction is essential to the successful deployment of a mobile ad-hoc network. Moreover the use of modules allows features that are difficult to integrate in the stack and/or very time consuming. The volumtous calculations can thus be done in parallel and do not block the communication stack.
The different layers gain access to the modules by means of the communications layer. Every communications layer has its own list of active modules, which is defined in the configuration file and initialized at startup.
One major difference between an asynchronous layer and a module is the fact that modules are not accessing the buffer of a lower layer to get new messages. They are receiving all messages concerning them by means of the dispatcher. Therefore they have to register themselves with the dispatcher (this is done automatically at bootstrap of the system and does not concern the programmer of modules). The dispatcher (itself an asynchronous layer of the communication stack) is calling the method deliverMessage() implemented in the module for every message destined to it.[4] The execution of the method deliverMessage() has to be as fast as possible because it uses resources of the communication stack. The processing of the messages should thus be done by the module itself. Therefore it is often useful that the modules extend the class Thread. In that way the processing of the messages can be done independently of the communication stack.
There are some constraints a class has to fulfill to become a module: the class has to implement the interface IModule.

2.2.1 IModule

The implementation of the interface IModule calls upon the following methods:

- public void deliverMessage(Message msg)
- public void initialize(IConfiguration config, CommunicationsLayer com)
- public void startup()

As already mentioned the method deliverMessage() is used by the dispatcher to forward messages to the modules. Because the execution time of the this method should be kept as short as possible, it is often enough to put the message in a local buffer and to call the thread of the module. A typical deliverMessage method could look something like Algorithm 2:

Algorithm 2 deliverMessage method for a module

```
public synchronized void deliverMessage(Message msg) {
    buffer.add(msg);
    notify();
}
```

The two other methods demanded by the interface IModule are used to prepare and start the module. As the names already tell, initialize() is called upon bootstrap of the system, startup() upon bootstrap of the module. For startup() it is in general enough to start the thread(s) used for the module. Initialize() on the other hand serves to do all the preparing necessary:

- initalization of local variables.
- definition of local parameters.

[4] At registration a module defines the type of messages it is interested in.

- lecture of configuration information from the configuration file by use of the IConfiguration interface provided to the method.[5]
- local storing of elements of the communication stack framework, as dispatcher, message pool, communication layer, etc...

3 The Implementations

3.1 Neighboring Module

The neighboring module is a simple module that allows the communication framework layers of a given node to retrieve information about their neighboring nodes [Figure 3]. The information exchange is based on the hello message principle. Periodically every module sends hello messages containing node information (nodeID, nodeName, etc...) to all neighboring nodes. Neighboring nodes are all mobile devices reachable by a simple one-hop broadcast. Theoretically this limitation can be extended by forwarding the messages via a routing mechanism in the nodes. The neighboring module is not only responsible for the emission of hello messages, but also for the processing of incoming ones. All information collected by means of hello messages are stored in a neighbor table, similar to a routing table. The node information is stored in the table until a entry expires timeout occurs. Any layer having access to the module (as mentioned above, the layers can gain access by means of the communications layer) can then make use of these facts.
Because the neighboring module is doing two different tasks, it implements two threads. The first thread is the main module thread already introduced in section 2.2, the second one is a supplementary thread which emits hello messages in regular intervals.
All constant values named above (hello message interval, expiration timeout, etc...) are defined in the configuration file in the section 'modules.Neighboring'. They are read by use of the method initialize() of the module. [5]

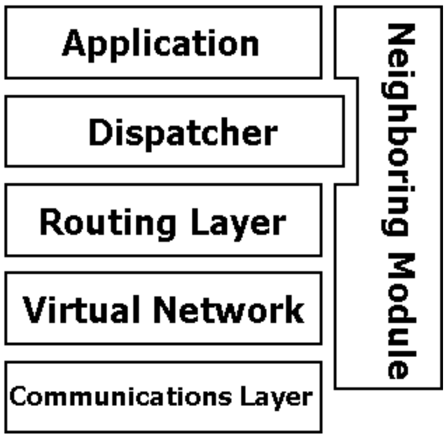

Figure 3: The communication stack with NeighborModule

How does the neighboring module work?

The package ch.epfl.lsr.adhoc.routing.neighboring consists of six classes:

- NeighborModule
- NeighborTable

[5]The file manet.config contains a section 'modules' where are the modules and its parameters are defined.

- NeighborTableEntry
- Hello
- HelloMsg
- HelloMsgFactory

The NeighborModule does all the processing of the hello messages (HelloMsg), thus everything that concerns the NeighborTable and its entries (NeighborTableEntry). In other words the module adds entries for new neighbors, updates and removes already existing entries. To do so the module implements the methods updateNeighborTable() and checkNeighborTable(). On the other hand everything that concerns the emitting of hello messages is done by another thread on an instance of the object Hello. Therefore the class Hello has to implement its own method public void run()!
Concerning the class HelloMsg it has to be remarked that it has to extend the class Message defined in the framework. Last but not least the class HelloMsgFactory is used to produce hello messages that are added to the message pool. The message factories are demanded by the framework and implement IMessageFactory. Because the factories are message specific, it is essential that for every new message type a new and corresponding factory is provided.

How to use the neighboring module?

The neighboring module stores all data in a neighbor table; the class NeighborTable extends java.util.Vector[6]. The elements of this vector are neighbor table entries (instances of the class NeighborTableEntry) which contain the necessary information about the neighboring nodes. These information are the id and the name of the nodes, as well as the time when the entry will expire. Any user (layer) can access these neighbor table entries by means of the getTable() method of the neighboring module which returns the module's neighbor table [see algorithm 3].

How to improve the neighboring module?

As for now the neighboring module only provides information about the identity of a node's neighbors. In no way this restriction has to hold in the future. As already mentioned earlier it would be very much possible to extend the use of the neighboring module by adding information about resources (energy, CPU, memory, etc...) at a given node (or any other fact interesting to a layer of the communication stack). Indeed the state of a node can very much influence the way a packet will take in the network, i.e. these facts could be very useful to a routing algorithm. Generally speaking: the neighboring module can be changed, extended or cloned as one wishes to do!

3.2 Statistics Layer and Module

The statistics layer is an element of the communication stack that collects information about incoming and outgoing messages of all types. The layer is a sniffer comparable to ethereal. All messages that pass through the communication stack are intercepted at the statistics layer and the collected facts are given to the statistics module which will process and present them to the user [Figure 4]. As for all modules, the user can be any layer of the stack (application, routing layer, etc...).

[6]NeighborTable extends Vector and not ArrayList or any other more sophisticated collection because of the limitations of the JVM.

Algorithm 3 Extraction of neighbor node information

```
NeighborModule nm;
...
NeighborTable nt = nm.getTable();
String ntStringRepresentation = nt.getString();
//This allows to represent the table in a graphical manner:
/*
 * Neighbor Table:
 * ————————-
 * ident1[name1] – 6245
 * ident2[name2] – 6906
 * ident3[name3] – 7802
 * ————————-
 */
NeighborTableEntry nte = null;
String singleNameString = null;
Enumeration enum = nt.elements();   //Remember that NeighborTable is a Vector
while (enum.hasMoreElements()) {
    nte = enum.nextElement();
    singleNameString = nte.getName();
    ...
}
//This allows to extract single values for all or several neighbors!
//Here the names of the nodes are searched.
```

3.2.1 Statistics Layer

The statistics layer is an asynchronous layer like e.g. the routing layer or the dispatcher - therefore it extends the abstract class AsynchronousLayer. This extension calls upon the implementation of two methods: initialize() and startup() [see section 2.1].
The other two methods that can be found in the statistics layer are handleMessage() and sendMessage(). Both of them are overriding their counterparts in the class AsynchronousLayer. Overriding these methods is necessary to collect the desired information, size and type, of the passing messages. These facts are included into a internal[7] StatMessage which is passed on to the dispatcher. As usual for modules, the dispatcher then forwards the message to the module by use of the deliverMessage() method. Example code and explanations for the two methods can be found in Algorithm 4.

3.2.2 Statistics Module

All the message information collected by the statistics layer is forwarded by the use of statistics messages to the StatModule [Figure 5]. The reader is referred to section 2.2 for all further information concerning the constraints on modules. As already seen for the neighboring module, the statistics module extends the class java.util.Thread to speed up the delivering of module specific messages (StatMessage). All the processing is done by the module itself and does not concern the communication stack.

How does the statistics module work?

The module contains a hash table in which the collected information is stored. The key object is a character representing the type of message, thus it has one entry per message type in the hash table. The value is a StatEntry which contains the following fields:

[7]StatMessages will never leave the communication stack. They are uniquely used for the communication between the StatLayer and its module.

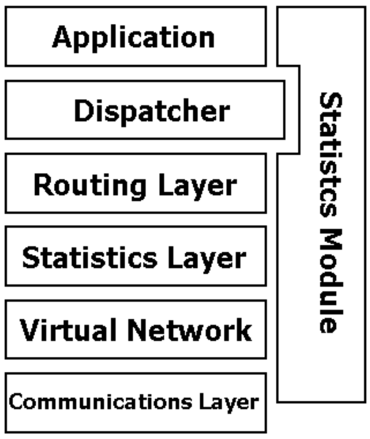

Figure 4: The communication stack with Statistics Layer and Module

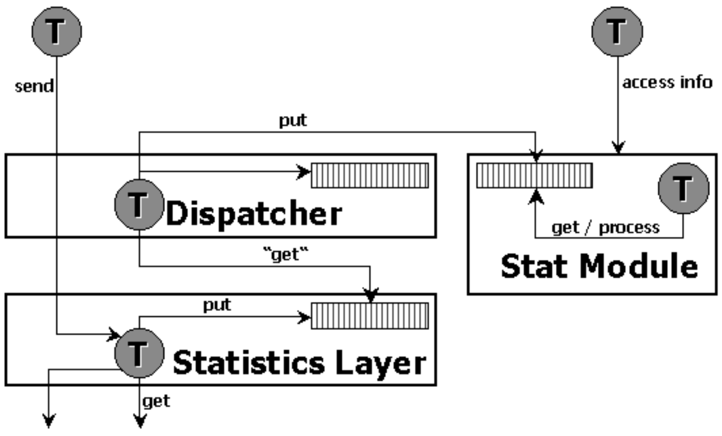

Figure 5: Message flow by use of the statistics facilities

- type [char]: the type of the message; same value as used for the key
- inSize [int]: the number of bytes entering the stack for this message type
- outSize [int]: the number of bytes leaving the stack for this message type
- in [int]: the number of messages of this type entering the stack
- out [int]: the number of messages of this type leaving the stack

Moreover there is a field startOfModule() which stores the time at which the method startup() was called. The value can e.g. be used to calculate message rates (e.g. msg/min).

How to use the statistics module?

The module implements four methods to retrieve the collected data:

- public Vector getMessageTypes()
- public Vector getTraficLoad(String msgTypeName)
- public Vector getNumberMessages(String msgTypeName)
- public String statsToString()

Algorithm 4 Overridden handleMessage() and sendMessage() for StatisticsLayer

```
public void handleMessage(Message msg) {

    int length = msg.getLength(); //get the size in bytes of the message msg
    //get a new StatMessage from the message pool
    StatMessage smsg = (StatMessage)mp.getMessage(statType);
    //include the type of msg and its size into the StatMessage
    smsg.setParameters(msg.getType(),length,true);
    super.handleMessage(msg); //pass the incoming message on to the upper layer
    super.handleMessage(smsg); //pass the statistics message on to the upper layer
}

public int sendMessage(Message msg) throws SendMessageFailedException {

    int length = super.sendMessage(msg); //pass the outgoing message on to the lower layer
    //get a new StatMessage from the message pool
    StatMessage smsg = (StatMessage)mp.getMessage(statType);
    //include the type of msg and its size into the StatMessage
    smsg.setParameters(msg.getType(),length,false);
    super.handleMessage(smsg); //pass the statistics message on to the upper layer (->dispatcher)
    return length;
}
```

The method getMessageTypes() returns a vector of strings containing all the names of messages contained in the hash table. These strings can then be used to retrieve further message type specific information.
The method getTraficLoad() returns the number of bytes passing through the stack for the message with name msgTypeName (e.g. Hello, StatMessage, etc...). The vector contains three values, all of them integers. At the first position the number of bytes leaving the stack (out-bound) are found. The second value contains the bytes arriving from outside (in-bound) and the third integer gives the total number of bytes (in-bound + out-bound) having passed through the statistics layer.
The method getNumberMessages() returns the number of messages passing through the stack. The vector returned contains also three integers in the same order (out-bound, in-bound, total).
The method statsToString() allows to retrieve directly a simple graphical representation of the statistics collected. This feature is comparable to the graphical choice provided for the neighboring module.
Algorithm 5 gives a little example of how these methods could be used for statistics retrieval.

How to improve the statistics module?

Because of their placement outside the normal layer hierarchy, modules know almost no limitations.[8] In that way a module is a very powerful tool to do extended calculations and time consuming processes.
The statistics module for example could be extended to provide information about the throughput or the latency of the system. In general one could use the module to do performance measurements of all kind.

[8] They are only limited by the available resources of the mobile device and not by the resources of the network (throughput, latency, etc...)

Algorithm 5 Statistics retrieval by use of StatModule

```
StatModule sm;
...
long startTime = sm.getStartOfModule(); //get the time of startup of the module
Date date = new Date();
Vector v1 = sm.getMessageTypes();
String msgName = (String)v1.elementAt(0);
long nowTime = date.getTime(); //get the time of statistics retrieval
Vector v2 = sm.getNumberMessages(msgName);
int out = ((Integer)v2.elementAt(0)).intValue();
double mpm = (double)out / ((double)(nowTime - startTime) / 60 * 1000);
return mpm; //returns the number of messages per minute...
```

4 Conclusion

The importance of interaction between the different layers of an ad-hoc communication stack demands a special infrastructure. Features that are in general not present in past telecommunication networks. The LSR-MANET communication framework provides an easy to use and straightforward implementation of a communication layer model. In addition to the asynchronous layer model, the stack independent modules provide a useful medium to increase the inter-layer communication. Modules, as they were presented in this report, collect and process data independently of the communication stack. Moreover, and that is important, they allow all layers to gain access to this facts.
The neighbor module can be used by the routing layer to collect information about the topology but also by the application to know its neighbors. In addition to the simple gathering of node information, this module could help to get facts about the batterie lifetime or other power management tools. It is no problem at all to add or extend existing modules. In this way the framework allows in a simple manner to know additional facts about the dynamic and ever-changing topology. The implementation of the statistics facilities gives a nice view on the possibilities the dynamic construction of the communication stack provides.

References

[1] S. Giordano, "Mobile Ad-Hoc Networks", pp. 1-4, 2000

[2] D. Cavin, "Wireless LANs, Mobile Ad-hoc Networks", p. 20, Nov. 2001

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

[4] U. Hunkeler, J. Bonny, "MANET Framework", sect. 2.3, February 2003

[5] U. Hunkeler, J. Bonny, "MANET Framework", sect. 3.4.3, February 2003