Semester Project

MANET Framework

Laboratory: LSR (Professor Schiper)
Advisors: David Cavin
Yoav Sasson
Project Team: Javier Bonny (javier.bonny@epfl.ch)
Urs Hunkeler (urs.hunkeler@epfl.ch)

Table of contents

Part 1: Project overview

1. Introduction

1.1. About the project

The MANET framework was a semester 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 at Lausanne (EPFL).

The project was realized by Javier Bonny (javier.bonny@epfl.ch) and Urs Hunkeler (urs.hunkeler@epfl.ch). Both are students in the third year of their telecommunications studies (http://ssc.epfl.ch).

1.2. About this document

This document is aimed at various kinds of audience. For this reason the document is structured into five different parts.

Part 1 ('Project Overview') gives general information on the project, its goals and the progress made. This part contains all information a classical project report should contain.

Part 2 ('Detailed Project Description') explains the internals of the project. This part should explain the solutions adopted without going into too much details.

Part 3 ('Programming Guidelines') is aimed at programmers who want to extend the project or use it for their purposes. It explains in detail how to write new extensions to the project and gives advice on critical points.

Part 4 ('References') contains a complete reference for the configuration file, as well as a list of standard extensions to the framework.

Part 5 ('Evolution of the Framework') gives advice to future maintainers of the framework about possible improvements and some ideas on how to implement them.

1.3. References to other projects

Other projects were based on this Framework. They implemented different parts (modules, routing algorithm). Here are the full references of those projects:

- *Use of Modules for better inter−layer communication*, February 2003

  by Reto Krummenacher (SSC5)

  reto.krummenacher@epfl.ch

- *AODV routing algorithm for multihop Ad Hoc Networks*, February 2003

  by Bertrand Grandgeorge (SSC3)

  bertrand.grandgeorge@epfl.ch

In this document, we will reference those project as Reto's and Bertrand's projects.

2. What is a MANET?

MANET stands for Mobile Adhoc NETwork. Think of it as the term for a reunion of mobile computers at a random place. For instance, people might form a MANET with their PDAs[1] at a train station. There is no central connecting point and no knowledge about the network configuration and topology. The mobile nodes[2] don't know each other beforehand. So how can they communicate?

This is the central question of MANETs. For the purpose of this project we assume that each node can send messages to all nodes within its radio range. Any node within this radio range might receive the message. The message might not be received by all nodes within the radio range due to interference. The sending of messages to all nodes within radio range is called broadcast.

In the train station example, lets assume that Alice has a PDA and stands 5 meters away from Bob who uses a laptop computer. When the PDA from Alice sends a broadcast message, the laptop computer from Bob might receive it. But the PDA from Cecil, who is standing 50 meters away, doesn't receive the message.

This example poses immediately a number of questions:

- How can Alice know about Bob (or rather, how can Alice's PDA know about Bob's laptop)?
- How could Alice send a message to Cecil?

These questions are handled under the terms of routing. Routing is the task of controlling the path of messages. For instance, if Alice's PDA finds out that there is another PDA within its radio range, lets say the laptop of Dahli, and that this laptop can reach the PDA of Cecil, then Alice could send a message to Dahli, which forwards the message to Cecil. And Cecil could answer by sending a message back to Dahli, who forwards it to Alice.

Discovering a possible route is probably the most challenging problem for MANETs, because:

- You don't have any beforehand knowledge about the topology of the network, and
- The topology changes over time

Of course you could just send a message to any node within your radio range and ask them to forward this message in turn to any node within their radio range, etc. This is called flooding. But as the name already implies, this is a waste of bandwidth and power (PDAs usually have a limited amount of power and want to send as few messages as possible).

Since there is no network structure guaranteed, nodes can only communicate through other nodes. This means that every node has to offer some public services, such as

1 Personal Digital Assistant: sophisticated electronic agenda or handheld computer
2 Here: any electronic device which can participate in a MANET (such as a PDA or laptop computer)

routing. There are projects analyzing the effects of egoistic nodes[3] (nodes that profit for instance from the routing offered by other nodes but don't do routing themselves). Such projects also propose mechanisms to force all nodes to participate (for instances, nodes might be excluded if they don't behave).

In order to help optimize and test routing algorithms and other network services (such as the mechanism to exclude egoistic nodes), we built a framework for MANETs, which permits easy implementation of a routing algorithm. Besides simulations on PCs the framework can be used on real devices, such as PDAs.

3 Stimulation Cooperationin Self−Organizing Mobile Ad Hoc Networks by Levente Buttyáne and Jean−Pierre Hubaux (http://lcawww/buttyan/publications/ButtyanH03monet.pdf)

3. What about JXTA for J2ME?

JXTA[4] is a P2P[5] framework. It permits the discovery of other computers running JXTA and exchanging messages between them. Furthermore it can forward messages through peers. This is especially used to connect a network protected by a firewall to other peers on the internet (supposing that one of the computers behind the firewall has somehow access to the internet).

There exists an implementation of JXTA for J2ME[6]. So why create an own framework? Why not simply use JXTA?

During the summer holidays we studied the internals of JXTA, its advantages and drawbacks, the current status and how it could be used. Finally we found several reasons why JXTA does not fulfill our needs:

- JXTA depends on infrastructure:
    - uses multicast for discovering peers (this is an integral part of JXTA)
    - gateways are statically configured
- JXTA assumes that the network topology doesn't change, or changes only slowly (when users connect/disconnect from the internet)
- JXTA isn't concerned much by performance (LANs and even the internet are relatively high bandwidth and low cost compared to the network technologies potentially used by MANETs)
- JXTAs main goal is to decentralize the server functionality (and thus create a new concept opposing the traditional client/server system)

Also J2ME only guaranties support for a single network protocol: HTTP. To use HTTP on J2ME, one must configure an HTTP gateway. This is because J2ME is aimed at cellphones, which cannot access a computer network directly. They usually make dial−up internet connections through an internet provider.

For these reasons we decided to build a completely new framework which is specialized on MANETs.

4 JXTA is a set of open, generalized peer−to−peer protocols that allow any computer on the network to communicate and collaborate.

5 Peer to peer: direct communication between any two computers on the same network (usually internet) without need for a server. There are no dedicated servers, but only peers. This new concept is said to replace the old concept of client/server networking.

6 Java 2 microedition: Java version for small devices (such as cellphones or PDAs)

4. Project goals

4.1. Assumptions

The whole MANET framework is based on the assumption that there is a mechanism whatsoever to send a message to all nodes within radio range. Routing tasks, such as sending a message to a specific node, are treated within the framework.

We decided to write the actual implementation of the framework in Java. The reason for this decision is that Java is extremely portable (machine and operating system independent). Furthermore Java has a good support for networking.

We decided to require Java 1.1.6 (or above), since this is the most recent version of Java that actually runs on Windows CE on iPaq[7]. We decided against supporting J2ME because it doesn't permit normal networking or JNI[8].

4.2. Tasks

The tasks for our project were:

- Define a framework: Define the different network layers (such as physical interface, routing, etc.) and the interfaces between them. Define the message format.
- Develop a sample implementation: Implement the framework in Java. Use multicast[9] as the physical layer.
- Implement flooding as a simple routing algorithm: No need for optimizing traffic, detecting routes or guaranteeing the delivery of messages.
- Implement a simple application, such as a Chat, for testing the framework

Initially, the following additional tasks were proposed:

- Create statistics
- Discover network topology

Later it was decided to migrate the task about statistics to Reto's project. The task of discovering the network topology became a whole semester project of itself.

We implemented the following additional tasks:

- Configuration file: The project became so complex that it was almost impossible to maintain without a configuration file. Our implementation permits, for instance, the definition of default values.
- Message pool: Right from the beginning we were concerned about the performance of the framework on small devices. One of the most time intensive

7 iPaq is a PDA from Compaq (now HP). We used iPaqs to test the framework.
8 Java Native Interface: permits to access subroutines written in an other language. This could be used for instance to access a special network card through a driver written in C.
9 Multicast is a special form of broadcast for tcp/ip, where a message is sent to all programs participating in a logical group

tasks in Java is to create new class instances. That is why we implemented a message pool where message objects can be recycled.

5. Personal goals

5.1. Why personal goals?

A semester project is a perfect opportunity to apply theoretical knowledge. As an additional challenge, our semester project was given to two persons.

We decided that we not only wanted to complete the tasks given to us, but that we actually wanted to learn something. We therefore decided to use a new development approach and to try to improve our team work.

5.2. Development approach

5.2.1. Cycles and phases

We decided to use a development approach called waterfall model. In this approach, one completely implements and tests a part of the final product before continuing with the next part. Each part we called cycle, and we divided each cycle into different phases: design, test implementation, code implementation, testing, documentation.

5.2.2. Design

Before any implementation phase we specify the goal of the development cycle. The classes are modeled using UML (with ArgoUML[10]). For every class and its methods the usage, expected behavior and constraints are defined.

5.2.3. Test Implementation

Before the actual classes are written, the tests for them are already written. This ensures that the tests are solely based on the specifications of the classes. The tests should test all the specified behavior of the classes, in particular the critical situations (bad arguments, etc.). We use a test approach called blackbox testing. In this approach, the tests don't have any knowledge about the internals of the objects to test, but can only use their public members to test conformity to the specification.

5.2.4. Code Implementation

During the code implementation, only the most important aspects or unusual solutions are documented.

5.2.5. Testing

After both the Test Implementation and Code Implementation phase have being completed, we run the tests generated in the Test Implementation phase. If those tests detect errors or non−conformities, we go back to the Code Implementation phase. Otherwise we proceed with the Documentation phase.

10 ArgoUML is OpenSource and written in Java (http://argouml.tigris.org/).

5.2.6. Documentation

Every class and every method is documented (JavaDoc[11]). In addition to the Java documentation, we write a report for every development cycle.

5.3. Team work

This special development approach makes it very easy to share the work. The design phase is completed in a meeting. One of us then implements the tests based on the design. The other implementsd the code. After both of us have finished their respective tasks, we make the testing together and go, if necessary, back to our previous tasks. Once the testing is completed successfully, one of us makes the JavaDoc comments and the other writes a report for the cycle.

11 JavaDoc is an utility that extracts special comments from the code and creates a webpage. This allows to have the documentation of the code directly inside the code, and still access it without having to search the source code. It also alows references to other parts of the code.

6. Design guidelines

6.1. Why design guidelines?

During the whole project we kept some guidelines in mind. This allowed us to have a more consistent implementation.

6.2. Performance

To improve the performance of the framework, we decided to minimize, wherever possible, the creation of class instances. The most extreme example of this is the message pool. Another example is the reception of messages, where we use the same buffer throughout a complete run of the framework.

Since the framework will be at the base of many projects, and not all programmers might be concerned about performance, it is even more important that the framework is as optimized as possible.

6.3. Modularization

It should be easy to change certain behavior aspects by just exchanging a class. This is especially necessary for a framework that should permit the development and testing of new algorithms. For instance, one cannot know at the time of programming which network layers and what types of messages will be used. But it is necessary to know the notion of layers and messages. We therefore created abstract classes, which can simply be implemented differently for different behavior. The actual classes are not hardcoded into the program, but are rather initialized from a configuration file.

6.4. Easy to use

The MANET framework is a fundament on which many other projects will be based. It is therefore necessary that the project be as easy to use and maintain as possible. Furthermore functionality that is likely to be used for other projects, or even other parts of the framework, should be implemented only once and be easy to access. Thus we have to implement, optimize and test those methods only once.

7. Conclusion

7.1. What works

The MANET framework is in a state that can be used for testing new alogrithmes and building implementations for real devices. There exists a sample implementation of a chat application which can broadcast messages to all nodes within a configurable number of hops[12]. It is also possible to specify a destination node. The framework is completely configurable through a configuration file.

We were surprised by the efficiency of the development approach. At first it seemed to take a bit longer. But already at the next cycle we had to change part of the implementation we made before, and we were really glad that we had the tests for the old code.

This approach permitted us to produce unusually stable code.

7.2. What was different than expected

Towards the end we were not able to stick strictly to our own development approach. The reason is that at the end the code became more and more dependent on other parts of the code. This made it virtually impossible to test a module just by itself. We replaced therefore the blackbox testing approach by a test method called code review. With code review, one writes the code and the other goes through it and tries to find all the mistakes and possible bugs. We think that this approach helped us to find bugs that could not have been found with blackbox testing and that it permitted us to further improve performance of the code.

7.3. Final statement

We are positively surprised by the outcome of the project. We think that we were able to exceed the initial goals and to produce an extremely stable and well designed implementation.

12 The number of hops is the maximum number of times a message is forwarded.

Part 2: Detailed project description

1. Message path

1.1. The principle of routing

Without routing a node would only be able to send messages to its direct neighbors. Routing allows a message to pass from one node to the next until it reaches its destination.

The simplest routing algorithm is flooding. Using flooding a node simply sends any messages it receives to all its neighbors. However, flooding is not very efficient (in terms of bandwidth and collisions). Therefore many research projects are going on to discover the ideal routing algorithm.

Another routing algorithm is AODV[13].

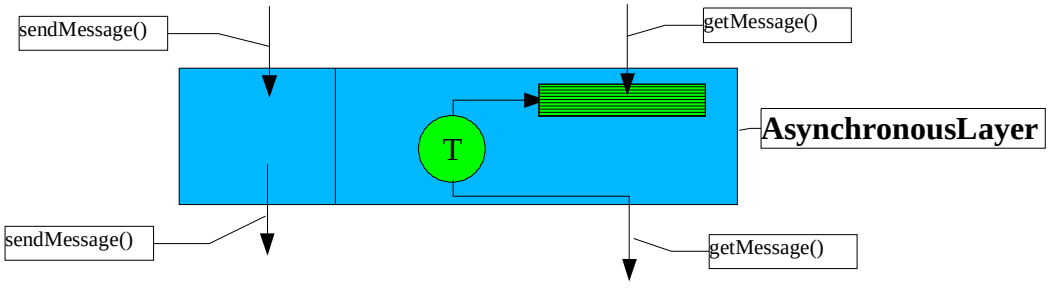
1.2. The principle of layers (class AsynchronousLayer)

After receiving the message on the physical interface (radio interface) until this message is delivered to the final destination (i.e. application), the message passes through several distinct layers. Each layer analyses the message and performs certain tasks. For instance there is a layer that transforms the binary representation of the message sent over the physical link into an object that can be handled more easily. Another layer just counts all the messages that were sent and received.

Some of these layers don't need to appear in a specific order. It might even be interesting to change the order of the layers. Furthermore it is not possible to foresee what kind of layer will be needed in the future.

Therefore we designed a unique interface between any layer. In fact, we implemented an asynchronous mechanism for receiving messages and a synchronous mechanism for sending them.

To simplify the implementation of a layer, we created an abstract layer, called AsynchronousLayer. This layer could be used as is. It then would simply pass messages in both direction without performing any operation on them.



The AsynchronousLayer has its own thread, which will always try to get the next message from the layer below, using the method getNetworkMessage(). If no message is available, the lower layer's getMessage() method will block until a new

13 Adhoc On−demand Distance Vector. see Bertrand's Project

message arrives. As soon as the layer gets a new message from below, it calls the handleMessage() method. This method simply puts the message into the layer's buffer. If the thread from the layer above was blocked because no message was available in this layer, the thread from above will be released.

When the sendMessage() of the AsynchronousLayer is invoked, it will invoke itself the sendMessage() method from the layer below to send a message.

To actually implement a layer one has only to extend the AsynchronousLayer class and to override one or more of the following methods: sendMessage(), getNetworkMessage() and handleMessage().

1.3. History of the different layers

As the first layer we implemented the network layer: AsyncMulticast. With this layer we were able to send messages onto the network and receive messages from it.

To demonstrate this, we then wrote a chat application. The chat application became a layer of its own only later.

Other layers might need some general utilities, such as a list of current direct neighbors. Such utilities are outside of the normal layer hierarchy. We therefore wrote a Dispatcher layer. The dispatcher knows about utilities (in the terminology of the framework called modules). The dispatcher filters all incoming messages. If it finds a message with a type that is used by a module, it forwards the message, directly to this module.

With this, Reto[14] was able to implement a module that maintains an up−to−date list of the direct neighbors. We enhanced the chat application to show the current list of such nodes.

Reto then added a statistics layer, which counts the number of messages and of bytes sent and received. This information is forwarded to a module. We enhanced again the chat application to show the current statistics as well.

In this state it was still not possible to test a routing algorithm because current radio network interface (such as 802.11b or BlueTooth[15]) don't support multihop adhoc networks, and on a wired ethernet network the notion of radio range doesn't make sense. We therefore added another layer which permitts to specify virtual networks. A virtual network simulates the connectivity between two nodes. With this it became possible to specify that a node could only send and receive a message on this or that virtual network. It was thus possible to limit the interaction of different nodes on the same physical network and to simulate an environment where multihop routing is necessary.

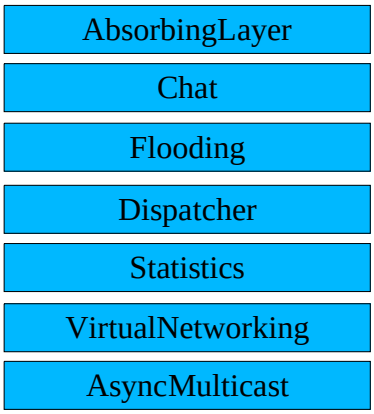
We then implemented the Flooding layer. This layer implements the routing algorithm called flooding. We further enhanced the chat application so it would be

14 See Reto's project

15 Actually, it is possible to build multi−hop networks with BlueTooth (this is called scatter net). But the organization of such scatter nets is not very flexible and therefore not usable to test general implementations of routing protocols.

possible to specify the destination.

Since it cannot be predicted what message types will be used and whether a particular node handles all message types, the last layer's buffer would just accumulate all messages with message types that are not handled by this node. To prevent this, we implemented a final layer, the absorbing layer, which just discards any messages that arrive. With this it is for example possible to build a routing node which just discards broadcast messages (which would normally be propagated through the routing layer further up the stack).



1.4. AsyncMulticast

1.4.1. Why multicast?

The initial idea was that we could use different multicast groups for different virtual networks and simulate with this a multihop environment. However it was impossible to implement this approach and keep the simple model of the layer stack. This is because you would need a separate thread per multicast group. This can either be done by completely changing the AsynchronousLayer class, or by changing the layer stack model to allow several instances of a layer to exist on the same level of the stack.

We therefore decided to use the VirtualNetworking layer instead. We still keep the multicast for accessing the wired network because it offers the same possibilities as the simpler broadcast. Additionally it would be possible to configure two different sets of nodes to use different multicast groups. Thus two different teams could test their enhancements to the framework on the same physical network without interfering.

1.4.2. How does it work?

The AsyncMulticast class overwrites the method sendMessage() and the method getNetworkMessage().

In sendMessage(), the message object is serialized (see *Part 2 chapter 2.2 Message* for details on how this is done) and then directly sent to the multicast group.

The method getNetworkMessage() tries to receive a multicast packet on the physical interface. When a packet is received, the AsyncMulticast uses the MessagePool (see *Part 2 chapter 2.1 MessagePool* for details) to create a message object and returns it.

1.5. VirtualNetworking

1.5.1. How does virtual networking work?

You can specify up to 20040 virtual networks (see also *Part 2 chapter 2.2.1 Network Bits*), which are identified by their number. Each message will contain information indicating on which virtual networks the message was sent. When the message is received by a node, the messages are filtered depending on whether the node can receive messages on one of the virtual networks on which the message was sent.

1.5.2. What happens when a message is sent?

The VirtualNetworkingLayer class overrides the sendMessage() method. This method simply adds a list of all virtual networks on which this node sends messages. It then passes the message to the layer below.

1.5.3. What happens when a message is received?

The VirtualNetworkingLayer class overrides the handleMessage() method. If the message was sent by this node, it simply puts the message into the buffer (a node always receives all messages it sent). Otherwise for each of the the virtual networks for this node the method tests whether the message was sent on that network. If this is the case, the method uses a random number generator to determine with a configurable probability for that particular network whether the message should be received. If so, the message will be put into the buffer, otherwise the method continues with the next virtual network. If the message could not be received on any of the virtual networks, it is simply discarded.

1.6. Statistics

1.6.1. How does the Statistics layer work?

The statistics layer counts all incoming and outgoing messages. It also counts the bytes needed for sending and receiving them (the bytes that were sent on the network).

The statistics layer does not maintain the statistics information itself, but it rather sends a message of its own to the statistics module. Those messages are never sent on the network, they are strictly kept inside the layer stack.

For details consult Reto’s project.

1.7. Dispatcher

1.7.1. What is a module?

A module is an implementation of an utility of general interest. Such a utility might be an up−to−date list of direct neighbors (see the neighboring module from Reto's project) or statistics about messages sent and received (see the statistics module from Reto's project).

Modules are not clearly placed within the layer hierarchy, but might be needed at any point within the framework. For this reason the dispatcher removes messages for modules from the layer stack and directly delivers them to the corresponding modules.

1.7.2. How does the Dispatcher work?

The dispatcher maintains a list of registered modules and the message type they are interested in. The Dispatcher class overrides the handleMessage() method. When a message arrives, this method checks whether the message's type is one of the types for which a module is registered. If this is the case, the dispatcher directly contacts the module and delivers the message (through the module's deliverMessage() method). Otherwise the dispatcher puts the message into its buffer.

1.8. Flooding

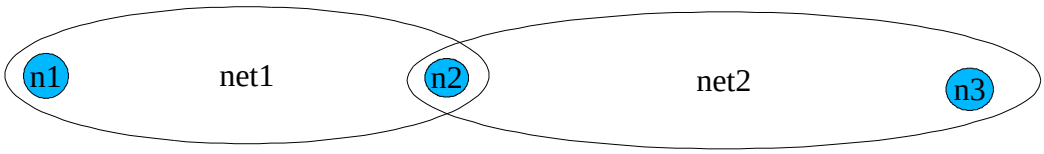
1.8.1. How does flooding work?

The Flooding class maintains a list of previously received messages. It overwrites the handleMessage() method. When a message arrives, this method tests whether the message has been previously received. If this is not the case then the method tests whether the message originated from this node. If this is not the case the method tests whether the message's destination is this node. If this is not the case the method tests whether the TTL[16] is bigger than or equal to 1. If this is the case, the method decides with a random number generator whether to retransmit the message. If this is the case, the TTL is decreased by one and the message is retransmitted after a random delay (through the DelayedTransmission class). In all other cases the message is not retransmitted.

If the message's destination is this node, or if it is a broadcast (destination is the number 0), then the message is put into the stack.

16 Time To Live: is decreased at every hop (retransmission). When it reaches 0, the message is discarded.

Below you see three nodes: n1, n2 and n3. Node n1 is on the virtual network 1, node n2 is on the virtual networks 1 and 2, and node n3 is on the virtual network 2. Node n1 cannot reach node n3 directly, but it can send a message to node n3 through node n2.



1.9. Chat

1.9.1. What does the Chat application do?

The chat application displays a window. In this window the user can write a short text message, chose a destination and then send the message. By default, the list of destinations only contains the broadcast address. For every message the Chat receives, it tests whether the originating node is already in the destination list, if not, it is added. So after receiving a message from a given node (that might have been sent as a broadcast), the user can choose that node as the destination for her/his message.

The Chat class overrides the method handleMessage(). If a message of a certain type arrives, the message's text message is extracted and displayed on the Chat window. Otherwise the message is put into the buffer.

1.10. AbsorbingLayer

1.10.1. Why an AbsorbingLayer?

If there are messages which are not removed from the layer stack by any of the layers, then these messages end up filling the last layer's buffer. This might happen if a node does not implement all layers or if there is a node on the network which implements a new or different layer or an other type of service (such as a module or application) which is unknown to this node. For instance, a node might use a minimalist implementation of the layer stack, which doesn't include the Chat layer. Such a node could be used as a router between other nodes running the Chat application. The messages for the Chat would then end up in the router's last layer.

To prevent an overflow of the last layer's buffer, the AbsorbingLayer was written, which will simply discard any arriving messages.

This approach was chosen to maintain the layer hierarchy as presented in this document. Thus it is not necessary to implement anything special to tell the last layer to discard messages, you can just put the AbsorbingLayer on top of the whole stack.

If there is an application, that is not implemented as a layer, then you one cannot use the AbsorbingLayer (otherwise the application doesn't receive any messages). In that case the application needs to treat unknown messages appropriately.

1.10.2. What does the AbsorbingLayer do?

The AbsorbingLayer overrides the handleMessage() method. All this method does is recycle the messages (using the message pool). By doing this, the buffer of the layer just below is steadily emptied.

2. Message handling mechanism (mhm)

2.1. MessagePool

2.1.1. Why a message pool?

In Java, one of the most time consuming operations is the creation of class instances. So throughout the whole project we aimed at avoiding unnecessary object creation.

Furthermore in the Java Specification the functionality of the garbage collector is not guaranteed. On a Java implementation for small devices, a good implementation of the Garbage collector could therefore easily be replaced by a poor implementation if this would reduce the code size of the JVM[17]. This could lead to memory leakage or temporary heavy use of memory (when objects are not freed fast enough).

Since message objects are heavily used, we decided to implement a message pool. So instead of just letting the garbage collector handle old message objects, we put them in a pool where they can be reused. And instead of generating a new instance of a message each time a message object is needed, we take already existing message objects and just reinitialize them.

2.1.2. How does the message pool create new messages?

At the beginning, and when need exceeds the capacity, the message pool needs to create new message objects. Since the message pool is a very general tool, it should be able to create and handle any type of message. To achieve this, there exists for each type of message a message factory (interface IMessageFactory). The message pool knows the message factory for each message type. If it needs to create a new instance of a message type, it simply uses the message factory for this type.

2.2. Message

2.2.1. The supertype for all messages

We decided to create a super class for all message types. This supertype groups information of global interest into a header and provides methods for serialization and deserialization of the primitive data types in Java.

The serialization and deserialization is implemented in the class Message to optimize performance. This was done because the objects and methods in the java api, that allow serialization, are optimized for writing to a stream (like a file or a tcp connection) rather than into a byte array (as is used on a broadcast network).

17 Java Virtual Machine: the program that emulates a Java processor

2.2.2. Header

Message Type

The ’message type’ header field starts at byte 0 and is 2 bytes long. This number should be interpreted as an unsigned integer (we use the data type *char* for this in Java). It contains a unique number for each message type. This allows the message pool to choose the right object or the right message factory from its pool to transform the byte array received by the physical network into the actual message object.

The message type header field is used by the message pool.

Version

The ’version’ header field starts at byte 2 and is 1 byte long. This number should be interpreted as an unsigned integer.

A future extension of the message format might changes the header information or the way in which data is serialized. When deserializing a network packet, incompatibilities can be detected or the network packet could be treated differently depending on the version number.

The ’version’ header field is used by the Message class for checking of compatibility of header information and the serialization mechansime.

Header length

The ’header length’ header field starts at byte 3 and is 2 bytes long. This number should be interpreted as an unsigned integer.

Even if header fields are added, an older implementation of the Message class should still be able to read all the previous header fields if the order of those fields doesn’t change. The older implementation will not be able to profit from the new header fields. But because the header length is known, the older implementation still can continue to read the actual message data.

The ’header length’ header field is used by the Message class to deserialize user data even if header fields have been added which are not known by the current implementation of the class Message.

Source Node ID

The ’source node ID’ header field starts at byte 5 and is 8 bytes long. This number is the unique identifier for the source node. This number is best interpreted as a long (even though how you treat it doesn’t matter). No assumptions what so ever should be made on the format of this ID.

The ’source node ID’ header field is primarily used by the routing layer.

Destination Node ID

The ’destination node ID’ header field starts at byte 13 and is 8 bytes long. This

number is the unique identifier for the destination node (the node to which this message finally should be sent). If this number is 0 (all bits are 0), then the message is a broadcast message. This number is best interpreted as a long (even though how you treat it doesn't matter). No assumptions what so ever should be made on the format of this ID.

The 'destination node ID' header field is primarily used by the routing layer.

TTL

The 'time to live' (TTL) header field starts at byte 21 and is 1 byte long. This number should be interpreted as an unsigned integer.

The TTL indicates, how many times the message can be forwarded. Every time this message is forwarded, the TTL value should be decreased by 1. If the TTL is 0, then the packet should not be forwarded.

The TTL header field is used by the routing layer.

Sequence number

The 'sequence number' header field starts at byte 22 and is 2 bytes long. This number should be interpreted as an unsigned integer.

Every time a node sends a message, the node initializes the message's sequence number with the current value and then increments its local sequence number. If the sequence number overflows (becomes bigger than 0xFFFF), then the sequence number starts again at 0.

The sequence number, together with the source node ID, uniquely identifies a message. Actually, because the sequence numbers can be restarted at 0 if an overflow occurs, this is not entirely true. However it is extremely unlikely that a message is still on the network when the original node sends a new message with the same sequence number (the maximum TTL is 255). Therefore the sequence number, together with the source node ID, is a good enough unique identifier for routing purposes.

The 'sequence number' header field is primarily used by the routing layer.

Network bits

Information about the network bits start at byte 24. The first byte (byte 24) should be interpreted as an unsigned integer that indicates how many bytes with virtual network information follow (see *Part 2 chapter 1.5 VirtualNetworking*).

There are at most 255 network bytes, thus limiting the maximum number of virtual networks to 255 * 8 = 2040.

The 'network bits' header field is primarily used by the virtual networking layer. It might be interesting to replace this layer by another layer that puts different information into this field (such as topographic information, see *Part 5 chapter 1.2 Geographical virtual networks*).

2.2.3. Serialization / Deserialization

A variable is serialized by adding a single byte to the output stream indicating the type of variable that follows, followed by the number of bytes needed to represent that value.

Integer numbers (int, short, long, char, byte) are serialized by extracting their bit representation with bit−wise operators. The order of the bytes is from most important to least important. This corresponds to the way serialization works in the Java API.

Floating point numbers (float and double) are serialized by transforming them to an integer representation (Float.floatToIntBits and Double.doubleToLongBits). The integer representation is then serialized as described above. The deserialization used the methods Double.longBitsToDouble() and Float.intBitsToFloat().

Strings are serialized by first adding a byte indicating the length of the string. This length should be interpreted as a unsigned integer. A value of 0 means the string is empty. A value of 255 (all bits set to one) means that the string is a null string. This limits the maximum length of a string to 254 characters.

The string is then transformed into a byte array using the string's getBytes() method. Note that this could lead to incompatibilities between two nodes that use different encodings, since no information about the current encoding is transmitted (the getBytes() method transforms the string into a byte representation for the native character set of the computer).

Boolean values have a special format, the value indicating the data type also directly indicates the value (there are two identifiers reserved for boolean).

Here is a list of all the data types that can be serialized. The list includes the identifiers used and the number of bytes the serialized data type occupies.

Datatype	***Identifier***	***length***
byte	b	1 + 1
char	c	1 + 2
short	s	1 + 2
int	i	1 + 4
long	l	1 + 8
float	f	1 + 4
double	d	1 + 8
String	S	1 + 1 + [0, 254]
boolean	0 / 1	1

3. Configuration file

3.1. Requirements for the configuration file reader

As the project grew more and more complex, it became apparent that some sort of dynamic configuration was unavoidable. As the project was already quite complex at that time, a flat configuration file (containing just data and no structure) was out of question. To anticipate future needs, we decided upon an XML[18] like format. But pure XML parsers are quit heavy and not yet included in the standard APIs for small devices. We therefore decided to create our own file format and implement our own parser. The file format can be read by an XML parser, however the parser cannot handle all aspects of XML. Thus the parser can later be replaced by a standard API XML parser.

3.2. Format

These are the restrictions of our own configuration file format in respect to XML:

- There may be only one tag per line.
- Only two tags are known: **section** and **param**.
- **section** expects exactly one argument: **name**.
- **section** may not be a single tag (i.e. it needs a separate closing tag).
- **param** expects exactly two arguments: **name** and **value**.
- **param** must be a single tag (i.e. must end with />).
- To use spaces and other special characters in a value, enclose them with double quotes (")
- Standard XML−comments are allowed, but they must stand on one or more lines by themselves.
- Comments (as long as they are on separate lines) and blank lines are ignored.
- There must be one principal tag. All other tags (excluding comments) must be inside the body of this tag.

18 eXtended Markup Language: A standard for creating human readable data files. This format is said to ensure that it can still be read in the future, even if the application that created the file doesn't exist anymore.

Here is an example of a file that could be read by the parser:

```
<-- Comment before main tag
 -- Second line of comment
 -->

<section name=MainTag>

  <-- One line comment -->
  <param name=anything value="some value"/>
  <section name=sub>

    <--
       More than one line comment
                                  -->
    <param name=otherthing value="something else"/>
    <param name=newParam value=123/>
  </section>
</section>

<-- End of sample file -->
```

3.3. Interface to access configuration information

3.3.1. IConfiguration

When the framework is started up, the configuration file is read. The file can then be accessed through the IConfiguration interface.

The configuration file is read only once, but there might be different instances of IConfiguration that access the same information. Each instance has its own search path. This makes it possible to specify different configurations for different instances of nodes, but at the same time specify default values.

A path is composed of a list of section names, each separated by a period. If the path denotes a parameter, then the last element is the name of the parameter rather than the name of a section. If the path starts with a period, then the path is taken as an absolute path, otherwise IConfiguration tries to locate the desired element by adding the path to each element in the search path list until it finds the element.

For example, the path ’.sub.newParam’ is an absolute path that points to the paramter containing the value ’123’.

The method addSearchPath() is used to add a search path. This path is always taken as an absolute path and may not start with a period. This new search path is added at the beginning of the list (and is thus searched first).

For example, if the list of search paths contains the entry ’sub’, then the path ’newParam’ would also point to the parameter containing the value ’123’.

3.3.2. Section

A Section object represents a section in the configuration file. It can be queried to retrieve subsections and parameter values. Thus it is possible to read a list of variable length, such as the list of message types. To do this one just retrives the section containing all the values of the list, and then reads all these values using the methods offered by the class Section.

Part 3: Programming guidelines

1. General programming information

1.1. The startup procedure of the framework

The MANET framework's principal class is ch.epfl.lsr.adhoc.Main. The main method requires at least one argument: the name of the configuration file. It further accepts a second argument: the name of the node. If the node's name is not specified, the configuration for the default node is used.

The syntax of the program invocation is as follows:

```
ch.epfl.lsr.adhoc.Main –c:<configuration file> [–n:<node name>]
```

There is no space between the colon and the argument.

Here is an example on how the framework could be started:

```
ch.epfl.lsr.adhoc.Main –c:manet.config –n:n1
```

With these arguments, the framework will use the configuration file called 'manet.config' in the current directory. It will further use the configuration for the node called 'n1'.

The framework itself is written in such a way as that it is possible to have more than one node running inside a single JVM. Each node is represented by an instance of the class CommunicationsLayer.

When the framework is started, the main method first reads the configuration file. It then creates an instance of CommunicationsLayer and passes a reference to the configuration information (IConfiguration).

The CommunicationsLayer first initializes a list of all known message factories and obtains an instance for each one (section 'MessageFactories'). It then reads a list of all known message types (section 'messageTypes') and registers each message type, for which a message factory exists, with the message pool.

The CommunicationsLayer then creates a list of all layers (section 'layerHierarchy') and obtains an instance for each one. It then initializes every layer, starting at the lowest layer (the network layer). The CommunicationsLayer also keeps a separate reference to the dispatcher layer (identified by the parameter 'dispNum').

The CommunicationsLayer then creates a list of all modules (section 'modules'), initializes them and registers them with the dispatcher.

Finally, all layers are started (method startup()), starting with the lowest layer. After that, all modules are started (method startup()).

1.2. The static class loader

Classes could be loaded dynamically (through the method Class.forName()). However we think that an implementation on J2ME should be possible and would open a whole range of very small devices. J2ME does not support reflection[19] and

19 package java.lang.reflection: An API for obtaining information about a class at runtime.

dynamic loading of classes. For this reason we implemented a class loader which knows already at compile time all the classes it will later be able to initialize.

So if you implement a new layer, module or message factory (or in fact anything that might be loaded dynamically), you should ensure that the class StaticClassLoader can instantiate a reference of your class. Simply add an if−statement (that tests for your class name) and a line that instantiates your class to the getInstanceFor() method.

1.3. Freeing messages

Message objects that are no longer used should be freed. That way the message pool can recycle them. To do this, simply call the message pool's freeMessag() method with the message to free as the argument.

Sometimes a reference to a message object is copied to different parts of the framework, which all will free the message object. To prevent a method from accidentaly freeing a message object that is still used somewhere else, the class Message counts the number of references. With Java 1.1.6 it is not possible to directly access the reference counters of the JVM. Therefore you have to increase this counter yourself when you give the reference to separate parts of the framework. You can do this with the method createCopy().

The createCopy() method is used by the Flooding layer, because a message can potentially be retransmitted and be sent further up the layer stack. This method is also used in the statistics layer.

1.4. Examples

To better understand this document, it's best to look at the code to see how it really is done. Here are some suggestions on which classes should best illustrate the explanations of this document. Also note that all core classes of the framework are well documented (you might want to use JavaDoc).

An example of a very simple layer is the AbsorbingLayer. Once you understand this layer, you might also want to look at the layer called Dispatcher. If you are interested in creating your own network layer, have a look at AsyncMulticast.

Probably the simplest message is TextMessage. Its message factory is the TextMessageFactory.

If you want to look at an example of a module, you should first look at the neighboring module. The statistics module is an example of how to collect information in a layer (StatisticsLayer) and analyze this information in a module. In particular, look at the way information is sent from the layer through messages to the module.

2. How to write a layer

2.1. The idea of a layer

A layer processes a message before it is sent to the network or after it has been received. There are various functions a layer can perform. For instance the virtual networking layer filters messages according to a header field.

A new layer has to extend AsynchronousLayer (or one of its subclasses). The super class already implements all the basic functionality of a layer, such as a thread which looks for new messages in the layer below and puts them into the buffer. However the subclass can change this default behavior.

2.2. Methods of interest

2.2.1. initialize()

The initialize() method is abstract and must be implemented. This method has access to the configuration and to the node's objects (such as MessagePool and Dispatcher). All variables should be initialized here.

Note that at the time of initialization, the layer stack is not completely initialized and no messages should be sent at that time.

2.2.2. startup()

The startup() method is abstract and must be implemented. Usually, this method simply starts the layer's thread by calling the start() method.

When this method is invoked, the layer stack is ready and messages can be sent. So if your layer sends periodic messages through its own thread, you can just start the thread and don't have to worry about delaying the sending of messages.

2.2.3. sendMessage()

The default implementation of this method just forwards the message to the layer below.

The lowest layer (the layer with access to the physical network: network layer) must override this method and send the message to the real network instead. Other layers might wish to override this method as well to modify some information of the message. When they are done, they should call the method super.sendMessage() to forward the message to the layer below.

2.2.4. handleMessage()

The handleMessage() method is invoked with a newly retrieved message from the layer below. The default implementation of this method just puts this message into the layer's buffer.

A layer might wish to override this method to modify some information of the

message. After this the layer should call super.handleMessage() to put the message into the layer's buffer.

2.2.5. getNetworkMessage()

The getNetworkMessage() method is invoked by the layer's thread to retrieve a new message. The default implementation invokes the getMessage() method from the layer below (which just returns the next message from the layer's buffer). If no message is available, this method should block until a message becomes available.

The lowest layer (the layer with access to the physical network: network layer) must override this method and return messages received from the physical network. There is no interest for other layers to override this method.

2.2.6. proposeNodeID()

Each layer can theoretically propose a unique ID for the node. Since this is most easily done with some information about the physical network (such as the MAC[20] address or the IP[21] address of the network adapter), the CommunicationsLayer class always asks the lowest layer to determine the node ID. Therefore the lowest layer must implement this method. There is no interest for other layers to implement this method.

Note that you should always use the method getNodeID() from CommunicationsLayer to obtain the unique node ID that is really used for this node.

The node ID is initialized bevore the layer's initialize() methods are called, so the layers can initialize a local copy of the node ID in the method initialize(). The lowest layer is an exception, since it is initialized before the CommunicationsLayer class initializes the node ID. Therefore the lowest layer (network layer) cannot access the node ID in the method initialize(), but has to wait until the method startup() is invoked.

2.3. How to configure the system

In the configuration file there is a section called 'layers' (usually inside another section called 'global'). There you add a section with your name for the new layer. This section has to contain at least one parameter: 'class'. This parameter's value is the fully qualified class name for your layer (including the package name). You can add other values that are used to configure your layer.

Once you added the layer to the configuration file, you can add it to the layer hierarchy. To do this, add a parameter to the section called 'layerHierarchy' (the default layer hierarchy section is inside 'nodes.default', but you could specify a different layer hierarchy for every instance of a node). The parameter names are the numbers of the layers in the order they appear in the stack, starting with zero for the lowest layer.

20 Media Access Control: the hardware address of network adapters.
21 Internet Protocol

Once you added the layer to the layer hierarchy, check that all layers in the hierarchy have consecutive numbers. Check that the value of the parameter called ’numLayer’ corresponds to the actual number of layers in the hierarchy. Also check that the value of the parameter called ’dispatcherNum’ corresponds to the number of the dispatcher layer.

3. How to implement a new message type

3.1. How to write a new Message type

A new message type extends the class Message or one of its subclasses. The superclass handles everything about the header and provides methods for the serialization of the data.

A message has some values it wants to transmit. It has therefore a number of internal variables plus the corresponding getter and setter methods. But these variables are not automatically serialized/deserialized when the message is sent/received. To do this, you have to implement the methods prepareData() for serialization and readData() for deserialization.

In the method prepareData(), all you have to do is to add the variables to the output stream using the corresponding addXXX() methods.

In the method readData(), all you have to do is to read the variables from the input stream using the corresponding getXXX() methods.

You must use the same order for serializing and deserializing the data.

You might want to overwrite the method reset() to reinitialize your variables when the message is recycled.

3.2. How to write a message factory

Message objects should not directly be created, but always be requested from the message pool. This way it is possible to recycle message objects and greatly improve performance. However the message pool still needs to create instances of your new message type. To do this, it needs a message factory for your message type.

A message factory is simply a class that implements the interface IMessageFactory. This interface defines a single method, createMessage(), which simply returns a new instance of that type of message.

3.3. How to configure the system

In the configuration file, there is a section called ’MessageTypes’ (usually in the section called ’global’). In this section you add a section for your message type. The name of the section is the numerical type of your message. This type has to be unique (no other message may use the same type), and the value must be between 0 and 65535 inclusive.

You must specify two parameters for this message type: ’name’ and ’class’. The ’name’ parameter contains a textual name for the message type (which has to be unique as well). Other configuration parameters should refer to the name rather than to the number (see for instance ’global.layers.Chat.msgType’).

The ’class’ parameter contains the fully qualified class name (including the package name) of your message.

Once you added the message type, you also need to add its message factory. To do this, add a section inside the section called ’MessageFactories’ (usually inside the ’global’ section). The name of the section is the name for the factory. This name need not be the same as the class name or the message type, but it must be unique.

Each message factory needs two parameters: ’class’ and ’msgClass’. The parameter ’class’ is the fully qualified class name (including the package) of the message factory. The ’msgClass’ parameter is the fully qualified class name of the message type this factory can create.

Note that the numerical message type has a similar function as the port number in IP. The same way it is possible to use the same application protocol (such as HTTP) on different ports at the same time, it is also possible to use the same message format for different message type numbers. Just create several entries in ’messageTypes’ with different names and numbers, but using the same class. You don’t need to repeat entries for the message factory, since the right factory for a message type is found by looking for a factory that can create instances of the class used for that message type.

4. How to implement a module

4.1. The idea of a module

A module provides information of general interest. A module must be able to send and receive data, but at the same time it should stand outside the normal layer hierarchy.

Modules implement the interface IModule. They are registered with the Dispatcher (a particular layer). When the dispatcher receives a message with a type for which a module is registered, the dispatcher immediately delivers the message to the corresponding module. Otherwise the dispatcher puts the message into the layer's buffer.

The dispatcher also has methods for obtaining a reference to a particular module.

4.2. Methods of interest

4.2.1. initialize()

The initialize() method has access to the configuration and to the node's objects (such as MessagePool and Dispatcher). All variables should be initialized here.

Note that at the time of initialization, the layer stack is not completely initialized and no messages should be sent at that time.

4.2.2. startup()

If the module has its own thread, it should be started here. When this method has been invoked, it is safe for the module to send messages.

4.2.3. deliverMessage()

The deliverMessage() method is called from the dispatcher to deliver messages. This method should return as fast as possible, because it otherwise blocks the main thread of the dispatcher. A typical implementation would for instance put the delivered message into a buffer, where it can be accessed from the module's thread for further treatment.

4.3. How to configure the system

In the configuration file there is a section called 'modules' (for instance in 'nodes.default'). In this section you add a new section for your module. The name of this section is the name you give the module. The name must be unique.

For each module, there are two parameters required: 'class' and 'msgType'. The parameter 'class' is the fully qualified class name of the module. The parameter 'msgType' is the name of the message type that the module handles.

You can add your own parameters there for configuring your module.

Part 4: References

1. ConfigFile

1.1. Structure

Here is the basic structure of the configuration file:

- searchpath
- global
  - MessageTypes
  - MessageFactories
  - layers
- nodes
  - default
    - modules
    - layerHierarchie
    - VirtualNetworks
  - ...

The section 'searchpath' determines how relative paths are treated.

The section 'global' contains information of general interest. Information in this section should not be changed in the section of a node. For instance all nodes should use the same message types.

The section 'nodes' contains the configuration information for the different nodes.

The section 'nodes.default' contains the configuration for the default node. Also if a value is not specified for a particular node (other than the default node), the value is taken from the default node.

The section 'nodes.<nodename>.modules' contains the list of modules that is used for this node. If this section is specified for a module (other than the default module) then only modules specified here are loaded for this node.

The section 'nodes.<nodename>.layerHierarchie' contains the layer hierarchy that is used for this node. The layers specified here are loaded and used in the given order for this node.

The section 'nodes.<nodename>.VirtualNetworks' specifies the virtual networks on which the node can send and receive messages. It also specifies the probability of reception of messages on the different virtual networks.

1.2. List of all the different parameters used and their meaning

1.2.1. Section searchpath

In this section, search path entries can be added. Search paths are used for relative paths (see *Part 2 chapter 3.3.1 IConfiguration* for more information on relative

paths). The names of the parameters are ignored. The order is the same as in the configuration file. Each entry is added at the first position of the search path (so the effective search path is in reverse order). The root path is always the last entry in the search path (that means that if everything else fails, a relative path is treated as an absolute path).

1.2.2. Section global

Section MessageTypes

Every message type has its own subsection here. The name of the subsection is the numerical message ID (must be unique). The subsections for the message types require two parameters: ’name’ and ’class’. The parameter ’name’ is an arbitrary chosen name (must be unique). The parameter ’class’ is the fully qualified class name of the class implementing the message type. This parameter is not required to be unique (i.e. two different message types may have the same implementing class).

Section MessageFactories

For every class that implements a message type, there must be a corresponding message factory. For each message factory there is a subsection here. The name of the subsection is an arbitrarily chosen name (must be unique). The subsections require two arguments: ’class’ and ’msgClass’. The parameter ’class’ is the fully qualified class name of the message factory. The parameter ’msgClass’ is the fully qualified class name of the class that implements a message (must be unique).

Section layers

General

For every layer there is a corresponding subsection in this section. The name of the subsection is an arbitrarily chosen name for the layer (must be unique). Each layer subsection requires at least one argument: ’class’. The parameter ’class’ is the fully qualified class name of the class that implements the layer.

Section AsyncMulticast

The AsyncMulticast layer requires three additional parameters: ’multicastgroup’, ’port’ and ’maxBufferSize’. The parameter ’multicastgroup’ is the multicast group that will be used for sending and receiving messages (a class D IP address). The parameter ’port’ is the IP port number used for sending and receiving network messages. The parameter ’maxBufferSize’ is the number of bytes allocated for receiving and sending messages. This is the maximum number of bytes a message may occupy on the network (the maximum packet size for sending/receiving over multicast).

The AsyncMulticast layer accepts one optional parameter: ’networkInterface’. This is the IP address of a network interface to use for sending and receiving messages. This is useful on computers which have more than one network adapter, since the multicast

socket is only bound to one interface. With this parameter you can determine to which interface the socket should be bound.

Section VirtualNetworks

The VirtualNetworks layer does not accept any additional parameters.

Section Statistics

The Statistics layer does not accept any additional parameters. See Reto's Project for more details.

Section Dispatcher

The Dispatcher layer does not accept any additional parameters.

Section Flooding

The Flooding layer requires three additional parameters: 'bufferLength', 'prT' and 'delayMax'. The parameter 'bufferLength' is the maximum number of messages that can be queued for retransmission. The parameter 'prT' is the probability that a message (that satisfies all other requirements) really is retransmitted. This must be a numerical, integer value between 0 and 100 inclusive (the probability in percent). The parameter 'delayMax' is the maximum delay between retransmission in milliseconds. That means, if the queue is not empty, at least every maxDelay milliseconds a message is sent (of course, Java gives no guarantees that the delay might not be longer).

Section AODV

The AODV layer requires a certain number of parameters. See Bertrand's Project for more details.

Section Chat

The Chat layer requires two additional parameters: 'ttl' and 'msgType'. The parameter 'ttl' is the initial time to live with which the Chat layer sends its messages. This value is a integer number between 0 and 255 (inclusive). A value of 0 means that the messages are only sent to the node's direct neighbors. The parameter 'msgType' is the name of the message type used for sending messages. The class that implements this message type must be the TextMessage class (or a subclass of TextMessage).

Section AbsorbingLayer

The AbsorbingLayer layer does not accept any additional parameters.

1.2.3. Section nodes

General

This node contains the configuration for every node. If no nodename is specified (see *Part 3 chapter 1.1*), the configuration for the default node is taken. Also, if a node

configuration does not include a particular value, then this value is taken from the default node. So in a way the default node is the general configuration. The different node configurations only change the values that are particular for that node.

Section modules

General

In this section, all modules are specified. Every module has its own subsection here. The name of this subsection is an arbitrarily chosen name (must be unique). Every module requires at least two parameters: ’class’ and ’msgType’. The parameter ’class’ is the fully qualified class name of the class that implements the module. The parameter ’msgType’ is the name of the message type this module uses.

Section Neighboring

The Neighboring module requires three additional parameters: ’entryExp’, ’allowedHelloLoss’ and ’helloInterval’. See Reto’s project for more details.

Section Statistics

The Statistics module does not accept any additional parameters. See Reto’s project for more details.

Section layerHierarchie

For each layer in the layer stack of a node, there is a corresponding parameter here. The name of the parameter is a integer number indication the layer’s position. The numbers are consecutive. The lowest layer (the network layer) has the number 0.

In addition there are two additional parameters: ’numLayer’ and ’dispatcherNum’. The parameter ’numLayer’ is the number of layers for this node. The parameter ’dispatcherNum’ is the number of the dispatcher layer.

Section VirtualNetworks

In this section there exists a parameter for each virtual network. The name of such a parameter is its network number. This is an integer number between 0 and 2039 inclusive (there are 255 * 8 = 2040 virtual networks). The value of the parameter is an integer number between 0 and 100 (inclusive). This is the percentage of the probability to actually receive a message that was sent on this virtual network. A value of 100 means that all such messages are received, a value of 0 means that all such messages are discarded.

Parameter daemon

The parameter ’daemon’ is directly inserted in the node’s section. This parameter is optional. It expects eigher "true" or "false" as value (the case of the letters is ignored).

If this parameter is set to "false", the thread of the last layer (usually AbsorbingLayer) is a normal thread instead of a daemon thread. This might be necessary for a node that

does not specify an application with a non−daemon thread. In Java an application stops execution as soon as no non−daemon threads exist anymore. This parameter might for instance be used for a router node with a very simple layer stack.

2. List of all known message types

2.1. RReq

Class: ch.epfl.lsr.adhoc.routing.aodv.RReq

Message Factory: ch.epfl.lsr.adhoc.routing.aodv.RReqMsgFactory

This message is used by AODV. See Bertrand's project for more details.

2.2. RRep

Class: ch.epfl.lsr.adhoc.routing.aodv.RRep

Message Factory: ch.epfl.lsr.adhoc.routing.aodv.RRepMsgFactory

This message is used by AODV. See Bertrand's project for more details.

2.3. RErr

Class: ch.epfl.lsr.adhoc.routing.aodv.RErr

Message Factory: ch.epfl.lsr.adhoc.routing.aodv.RErrMsgFactory

This message is used by AODV. See Bertrand's project for more details.

2.4. StatMessage

Class: ch.epfl.lsr.adhoc.statistics.StatMessage

Message Factory: ch.epfl.lsr.adhoc.statistics.StatMessageFactory

This message is used by the statistics layer. The statistics layer sends such messages, which are then routed to the statistics module. These messages stay inside the layer stack and should not be sent over the network.

See Reto's project for more details.

2.5. Hello

Class: ch.epfl.lsr.adhoc.routing.neighboring.HelloMsg

Message Factory: ch.epfl.lsr.adhoc.routing.neighboring.HelloMsgFactory

This message is used by the neighbors module. The neighbor module sends such messages periodically to all its direct neighbors (ttl is 0). When the neighbors module receives such messages from a node, it knows that this node is a direct neighbor and updates its tables accordingly.

See Reto's project for more details.

2.6. Text

Class: ch.epfl.lsr.adhoc.chat.TextMessage

Message Factory: ch.epfl.lsr.adhoc.chat.TextMessageFactory

This message is used by the chat to exchange text messages.

Part 5: Future Evolution of the Framework

1. Extensions to the framework

1.1. Long Messages

At the moment, the maximum size of a message sent over the network is limited to the buffer size reserved for the AsynchronousLayer. The routing layer could implement a mechanism to split long messages into shorter ones (similar to tcp).

1.2. Geographical virtual networks

The VirtualNetworks layer creates virtual networks. But it is not possible to realisticicly simulate distance or interference. One could send coordinates instead of network numbers with the message. Then it would be possible to calculate the probability of reception as a function of the distance between the two nodes.

Additionally it would be possible to maintain information about noise level (basically a moving average function of the traffic and the distance from the emitting nodes). This could further modify the probability of reception (and make this probability dependent on the actual traffic in the air, etc.).

1.3. Query for unknown message types

When a node receives a message with a type it doesn't know, it could send a query to the source node and ask it about this type. Maybe it would even be possible to send a small program back which the user then could install.

Imagine the scenario where passengers meet in a train station and want to exchange data. It would be very annoying if they used different modules. But if the computer that doesn't understand the messages it received, just could ask the sender to also send the program (or a demo version of that program, or just a viewer to display the data), then they could still communicate.

1.4. Tests

One could develop a detailed test suite for this framework. For instance, it should be possible to check whether messages are correctly freed, using Java 2 and the java.lang.ref api. The tests don't need to be compatible with JDK 1.1.6. If the tests succeed under Java 2 and the framework compiles with the JDK 1.1.6, then it should work properly under JDK 1.1.6.

1.5. Searchpath

At the moment, there is no mechanism to ensure the order of the searchpath configured in the section 'searchpath' of the configuration file. The current implementation keeps the same order it receives the information from the configuration file. But XML does not guarantee a particular order. Therefore there should be a mechanism that sorts the searchpath according to some criteria (such as its name).

1.6. Configuration hierarchy

It might also be of interest to create a policy on which values can be overridden at what stage and how this is done exactly.

At the moment, in order to change certain values for a node, the whole section containting these values has to be configured for that node. It has to be analyzed whether this is really necessary or how this could be solved alternatively.

1.7. Dynamic configuration

It might be interesting to be able to change certain configuration options on the fly. This could be especially interesting for the virtual networks layer. That way it would be possible to simulate node movements.