UNIVERSITY OF OSLO
Department of Informatics

Information sharing in mobile ad-hoc networks using a global metadata manager

Aslak Johannessen

April 30, 2008

Abstract

Digital information sharing is vital to improve efficiency in rescue and emergency operations. In the absence of infrastructure, the devices brought into the area by rescue personnel, might be used to set up a mobile ad-hoc network. This application domain is dominated by frequent merges and departures of nodes, resulting in an unstable and ever changing topology. Information sharing can be handled by using metadata describing instances of tables in the network. We disseminate metadata to all nodes, so that each has its own local structure containting all metadata available. The structure enables nodes to search for table replicas, and query these over delay tolerant paths. Since metadata is small in size the structure will remain small even with many tables existing, and storing it on every device is feasible. The characteristics of mobile ad-hoc networks makes the task of disseminating metadata across all nodes difficult, in particular in the presence of scarce resources. We have looked into research coming from the domain of ad-hoc networks to reuse techniques applicable to information sharing. The techniques come mainly from routing in delay-tolerant networks, and routing protocols of dense networks.

We have implemented three protocols which use different approaches to information sharing. The simple protocol uses a basic epidemic routing approach, propagating information from neighbor to neighbor. By using the broadcast protocol, we utilize the shared medium of wireless communication and its ability to send messages from one to many nodes, in addition to letting each node stop the synchronization if no metadata is needed, thus removing some of the redundant propagation paths. The semantic protocol utilizes the concept of groups amongst the nodes, by giving these first priority. If nodes of the same group are spread out through the network they will serve as multiple starting points for the following normal dissemination.

By taking part in the implementation of the MIDAS middleware, real life scenarios have been acquired, and tested using the protocols developed in this thesis. MIDAS is a large reasearch project aiming to produce a middleware that speeds up MANET application development. By testing the implemented protocols in NEMAN, an emulator environment, we have measured both bandwidth usage and performance. The implemented broadcast protocol is measured to use 50 times less bandwidth than the simple implementation. Further the concept of groups of nodes is utilized to give certain nodes higher priority in the dissemination process, which results in multiple starting points for the normal dissemination techniques. This protocol has only shown that the group priority is working, however more research is needed to verify that the multiple starting points give an advantage to the global dissemination. As we can see from the results and the techniques used, routing principles in mobile ad-hoc networks can easily be applied to information sharing showing good performance optimization, resulting in fast dissemination and high availability of metadata information amoung the rescue personnel's devices.

Acknowledgements

I would like to thank my supervisor Ellen Munthe-Kaas for her excellent supervision and guidance. Special thanks also to Odd Aurmo for helping me out with my English. I would also like to thank Matija Puzar and Katrine Stemland Skjelsvik for their cooperation during the implementation phase.

Aslak Johannessen
Department of informatics UIO
30 April 2008

Contents

List of Figures

List of Tables

List of abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
AODV	Ad hoc On Demand Distance Vector
API	Application Program Interface
CRT	Communication and routing
CTS	Clear to send
DB	Database
DDB	Distributed database
DDBMS	Distributed Database Management System
DENS	Distributed event notification system
DMMS	Distributed multimedia systems (Distribuerte multimedia systemer, Norwegian)
DS	Data synchronizer
EU	European Union
ERS	Expanding ring search
GMDM	Global metadata manager
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile communication
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
IFI	Institute of informatics (Institutt for informatikk, Norwegian)
IP	Internet Protocol
ISC	Information store count
LAR	Location-Aided Routing
LS	Local storage
MANET	Mobile ad-hoc network
MDS	MIDAS data space
MF	Message ferry
MIDAS	Middleware platform for developing and deploying advanced mobile services
NAM	Network Animator
NEMAN	Network emulator for mobile ad-hoc networks
NS2	The Network Simulator
OLSR	Optimized Link State Routing protocol
OLSRD	OLSR daemon
PROPHET	Probabilistic Routing
QA	Query analyzer
RREP	Route reply
RREQ	Route request
RTS	Request to send
SM	Subscription manager
SQL	Structured Query Language
TCP	Transmission Control Protocol
TTL	Time to live
UDP	User Datagram Protocol
UML	Unified Modelling Language

Chapter 1

Introduction

The focus for this thesis is rescue scenarios, and giving rescue personnel the ability to use devices aided information sharing. By exchanging information between both personnel and between rescue departments information can be at the correct place at the right time, and decisions being done with the correct knowledge. For devices to communicate a computer network will be required. The devices can use cell phone networks whenever available, however in many rescue scenarios such networks are not available, in these cases only one option remains, to use the devices as the network itself. The devices will have to form an ad-hoc network, cooperating on network tasks, using each other's connectivity. In such dramatically altered networks, other requirements will be governing the application than that of a traditional environment. These requirements are specific to the ad-hoc network, and embed asynchronous communication, spontaneous joins and departure of device's and delay tolerant communication. For an application to work over different networks a middleware has to be developed. By using this middleware an application can move between infrastructure, cell phone, and ad-hoc networks transparently. Operation in the ad-hoc network environment will become easier and more applications can be developed as the knowledge required for development improves. We will in this thesis look into how to make the information residing on one device known to all others, by using metadata. The dissemination of metadata is the core of information sharing, thus dissemination protocols need to be efficient both with respect to bandwidth and time usage.

1.1 Background and motivation

The mobile industry is making smaller and smaller devices with better and better communication features. The method of communication is no longer restricted to wires, nor application to using voice. Data transfer and data sharing is feasible on most mobile devices today. The hardware already is out there, and we have only seen the start of software utilizing this new hardware.

Mobile devices are often equipped with Bluetooth, GSM and Wifi. Traditionally these technologies have been used to connect to already established infrastructure, like GSM carriers or Wifi hotspots. The hardware does not limit the use to infrastructure, the software does. Both Bluetooth and wifi standards enables connectivity directly from device to device. This ability is called ad-hoc networking, and is one of the fastest ways of establishing networks. When a rescue organization, for example, arrives at a rescue site their efforts need to be organized, this can be done through an ad-hoc network. While the network does no longer rely on existing infrastructure, only the devices themselves supplying the organization with a network wherever they need it. This is extremely convenient when organizing joint fire, police and ambulance operations in remote locations as well as in urban areas. In remote locations infrastructure does not exist, in urban areas, inside buildings, or tunnels existing infrastructure may be blocked. In these situations there exists no alternative but to use existing nodes to form the network, hence ad-hoc networks.

By forming such networks we can efficiently become independent of infrastructure, thus only using the ad-hoc community to support the network. Many new consumer devices come with this network feature as one of the standard ways of connection.

Networks in the ad-hoc category are different to normal networks. A community of devices *is* the network, and all devices are participating in routing packets and other network services. This makes each device part of the "backbone" at the same time as they are normal end users devices. This scheme requires that each device contributes to the network by running a set of services to support the ad-hoc network.

When devices running cooperating services meet they will form an ad-hoc network. This network can deliver the same features as one would expect from a normal network. While using an ad-hoc network as a normal network, the dynamic of the network will not be utilized. New neighbors come and go, to the frustration of normal applications. Applications can use dynamical connectivity of the ad-hoc network to bring more relevant information to the user. Applications can stay connected in places where they earlier could not. Collaboration extends beyond connectivity, to features like information sharing, utility sharing and context information to mention the most widely used.

The nature of the ad-hoc network makes normal application development hard, as it has to take into account delay tolerant networks and frequent joins and partitionings. Portability is another issue, each application wants to work over all network types and be able to utilize gateways to internet connections when such are available. To make the application development feasible and portable, middleware software is needed. The middleware takes care of all the hard and difficult problems of ad-hoc networking and, ideally only showing off new and exciting features.

The vision of the ad-hoc network is to connect mobile devices, the software developed for these networks can give another dimension to how we use our devices. An example; Currently a system for car safety is being developed, this uses the car as a mobile device equipped with a radio communication, it will warn drivers along the road of hazards or accident. This can be done by letting information jump from car to car. When an airbag is triggered the car warns the cars around about the accidents. As the information reaches new cars, drivers will be aware of the accident so they can stop in time. With the support of an appropriate middleware, such applications in the future will be faster to develop and more innovative in their use of the ad-hoc networks.

Information is not only limited to events, like in the car example, but also to information gathered or stored at any node. In rescue and military scenarios this can, for example, be maps, instructions or registers of patients. This type of information is physically residing in one location, but is useful elsewhere in the network. For information to be available to each and every device, efficient protocols for information sharing must be developed.

1.2 Application domain

Rescue like scenarios are characterized by having;

- devices carried into the area by rescue personnel
- frequent arrivals and departures of devices
- little or no infrastructure

The connectivity among the nodes is often sparse, meaning that a large area is covered only with a small number of nodes, this leads to frequent network partitions.

Take a train crash for example, where a train has collided inside a tunnel; connectivity to the rescue personnel inside the tunnel can only be forwarded through the other rescue personnel on the ground, often relying on only single devices to transfer information from inside the tunnel to the outside. On the outside, large groups of devices are gathered waiting for patients or coordinating the efforts. To coordinate such a scenario information availability is crucial, thus the information has to be made available to the rescue personnel which needs it. Both areas with few devices and areas with many, must be handled, so that the network service and information sharing is optimal. Information gathered inside the tunnel has to be made available to the coordinators or ambulance personnel outside.

This availability has to be done even though there is at the moment no connection into the information gathering for the rescue worker, this will be achieved via delay tolerant networks, using store and forward techniques. This situation can arise when the network is sparse, where each node has a small number of neighbors and many partitions exist. By using the mobility of each node, we can still get connectivity and information sharing amongst the nodes. By using these delay tolerant concepts, we will store information on each node as it gathers it. Upon meeting a new node we propagate the information to the new node, hence forward. By doing this a fireman can walk from inside the tunnel, and store or carry the information outside to the awaiting rescue personnel. When the fireman is walking back into the tunnel new information from the outside can be carried inside. By this store and forward technique metadata, along with requests for the actual data, can be transferred between devices with no apparent connection, thus handling the frequent partitioning and remerger, normal to the rescue scenario. This can solve the problem of unconnected partitions, but introduces many challenges that are to be handled, thus turning all "quality of service" terms on their head.

The domain can also be partitioned, the mentioned police, fire and ambulance departments have to cooperate to perform a successful rescue operation. The different devices coming from the different departments have to work together, regardless of which department they are under or type of device they use.

This cooperation will be handled by using the middleware, so that each device can relate to a unified middleware which transparently hides the differences.

Development in the ad-hoc environment has been ongoing for some time, mostly concerned with routing in ad-hoc networks. This has produced a large quantity of routing protocols handling any scenario from dense via sparse to delay tolerant. This thesis will try to use the large researched area of routing protocols in ad-hoc networks to enhance information sharing. By using the principles of routing, sharing of information will be done in an efficient manner, utilizing the dynamic nature of the network. By implementing this strategy I will show that today's devices are capable of using ad-hoc network technology to build dynamic information data storage, that expands as devices meet devices. While talking about small devices and radio communication, we will not have large storage space nor communication lines. In fact both assets will be small and slow, which give implications into how we can accomplish this task. The solution we are going to explore is the knowledge of knowledge, metadata. By not moving the entire knowledge around, we distill the metadata information and make a small piece of data called metadata. This metadata is then disseminated through the network. This cooperation between devices will take advantage of the mobility of each device. By exchanging information each device will be able to discover information outside itself and its network. Through this approach information that would forever stay hidden is discovered. This is accomplished through an advanced device community, the network, without any network carriers or infrastructure. Which gives these networks and devices the ability to survive and function in disaster areas or other areas where infrastructure is not established.

1.3 Problem description

I have set out to handle information sharing in ad-hoc networks. By networks I include both dense, sparse and delay tolerant networks. Information sharing is the ability for each node to share its information with the ad-hoc network and for each node to be aware of all other information in the network. More formally;

- 100% dissemination. All devices in the entire ad-hoc network are required to know of all metadata. Information is to propagate to all nodes in its partition, in addition to all partitions that can be reached in a delay tolerant manner.

- Global view. All nodes require all information. This demands that no nodes will be starved with respect to any metadata.

These requirements are absolute, without meeting them we can not say that information sharing occurs. If some areas of a partition or an ad-hoc network do not know of a metadata element, which is known to others, the information sharing will have failed. To avoid this situation, we enforce rules to eagerly propagate information to neighbors. The triggering events must be established so that bandwidth and processing are spared, but still all demands are met.

In addition to the absolute demands will we try to optimize the protocols so that they;

- Use as little bandwidth as possible.
- The dissemination goes as fast as possible.
- Queries have low response time.

When the absolute demands are in place the focus will shift to get the protocols running as cost efficient as possible, along with having the performance as high as possible. While existing in the ad-hoc networks where bandwidth and processing power are key resources, these requirements will be important to the performance of the entire network. If one part of the middleware uses more than it needs this resource is taken away from the applications, resulting in degradation of application potential and performance.

To optimize for performance and meet the demands put upon the protocols we will use principles already developed for routing in ad-hoc networks. To limit the load on the protocols only metadata will be disseminated, data must be activly requested. The problem description is summarized in the following quote;

> *Develop an efficient information sharing protocol, using principles from ad-hoc network routing protocols.*

1.4 MIDAS

MIDAS [15] is a middleware set out to make the life easier for the application developer. Making the use of ad-hoc networks easy and with less complexity than the situation is today. MIDAS will serve as a middleware that each application can connect to. It delivers services as send and receive, context routing and even has it own data space for storing and retrieval of information. It is in this data space, called MDS (MIDAS data space), the information sharing is taking place.

The work in this thesis was carried out as part of the MIDAS project. MIDAS is an EU project, consisting of different parties along with the University of Oslo.

1.5 Outline

The rest of the thesis is organized as follows;

Chapter 2 provides background material. This is done since the following chapters are built upon this background knowledge of wifi and ad-hoc network environments and standards.

Chapter 3 introduces the information sharing approach used in MIDAS. How best to do dissemination of metadata with the purpose of information sharing is discussed.

Chapter 4 discusses how to handle the problems encountered in ad-hoc networks and wifi environments when doing information sharing, and an in depth discussion into how the problems should be managed. A further specification of the problem description is also included.

Chapter 5 includes design decisions on how to implement metadata sharing. This is done through explaining each of the protocols and their messages.

Chapter 6 handles the implementation. In this chapter, the detailed implementation is discussed and specified. Both the implementation of the protocols and the surrounding framework are described. Development techniques and practises are also introduced.

Chapter 7 includes a specification of how to test the implementation specified, laying out the test scenarios and the test setup, describing the control and logging mechanisms.

Chapter 8 contains the test results and the evaluation. In detail description of the performance, with respect to the absolute performance requirements but also evaluation of protocol against protocol is done.

Chapter 9 contains a conclusion and possible future work.

The appendix is divided into 5 parts, describing the cd appended to this thesis and some important source code files. Uml documents central parts of the implementation are included, before a re-execution recipe is described.

Chapter 2

Background

This chapter gives the background that is required when moveing on to the discussion and implementation chapters. The background consists of introduction to the physical enviroment, the MANET and wifi technologys including the standards that are used. The enviroment is inhereting problems due to physical constrants, these are descirbed and solutions studied. Further is the transport layer routing protocols introuced for diffrent types of topologys. The MIDAS middleware is introduced by describing its vision, goals and sub components, going into detail in MDS. Before we talk about the database asspect of MDS consering this thesis. Simulation and emulation envirioments used to test comunication application of all sorts are descirbed in Section 2.7. The chapter is finishing of with a description of the Ad-hoc Infoware project which MIDAS is building on, in particular intrest is the DENS network.

2.1 MANET characteristics

MANET is an acronym for mobile ad-hoc network, which is a self-configuring wireless ad-hoc network [6]. The topology of the network is defined by the connectivity of all the nodes at any given time. The nodes are not limited to running routing protocols but will also run user applications and services. The nodes are free to move as they wish, this makes for rapid changes in the topology. Nodes can disappear and reappear, due to both movement and shutdown, without notice. Such events are referred to as network partitioning and merging. During both partition and merge a node can alter its set of neighbors which are defined to nodes that are within reach. We will refer to these nodes as neighbors or one hop neighbors; hence they are one transmission range away. Nodes that are one hop further away are called 2 hop neighbors since they have a two hop route to the node. Such multihop routes can involve unstable connectivity of intermediate nodes. This is handled differently by different routing protocol classes. We will look into these protocols shortly.

2.1.1 Connectivity characteristics

In a dense network of nodes, the connectivity at each node is quite good since each node reaches all other nodes via multihop routes. By connectivity at a point in time we require node A to have connection with node B. If node B is far away from node A, it can use a multihop route to B. This can be done if A has connectivity to some intermediate node that has connectivity, directly or through other intermediate nodes, to B. If there exists a chain of hops that leads from A to B, we call the connectivity a space path or a multihop route. This means that there is a route from A to B that is "stable" so that multiple packets can take this route. These space paths are frequent in dense networks and are part of its characteristics. We distinguish between routes that have intermediate nodes from those that are in direct reach of each other. The ones with intermediate nodes are called multi hop routes, and the direct ones one hop routes.

When there is no path from A to B at a given time, there is no space path. In some networks where no space path exist, mobility of the nodes can make it possible to send a packet from A to B, by using an intermediate node that later in time passes by B and delivers the message. We call such paths over time, space/time paths. Such space/time paths are part of the characteristics of sparse networks. In these situations normal routing protocols do not work. We need to use delay tolerant protocols which store and forward the packets.

To be able to route network packets over delay tolerant paths one would have to use other mechanisms than over space paths. In these cases the routing protocol would take into account, not only the current connectivity, but also the future connectivity. Mobility is an important part of the delay tolerant routing protocols, since it defines the connectivity. Routing under these conditions is done by placing the network packet at nodes that can lead to propagation of the packet nearer to the recipient. The selection of the store-and-forward nodes is the major concern for the different protocols.

Let's look at one delay tolerant routing protocol; where A and B are non-mobile nodes that are out of reach of each other and F is a node that frequently passes both A and B. We then have a space/time path between A and B. The only way to send packets from A to B is via F when it passes by the nodes. In turn F needs to retransmit the packet to the receiver when it reaches it. F then needs to store and forward the packet from the time it receives it from A to it can pass it on to B. We often refer to this situation as a message ferry while F is ferrying packets between the two nodes that are out of reach. The technique of message ferrying might seem like a universally good idea, but in practise it requires some knowledge of the movement pattern of the nodes. In most examples the ferry is a designated node with this one intent; the ferry is moving in a pattern where it is supposed to encounter the nodes in the network, it gathers messages like a postman. Ferry techniques differ from the ferry seeking out each and every node to the nodes actively meeting the ferry on its route. This pre-planned scheme is not feasible in general uncontrolled networks, thus different techniques need to be used. [22]

2.1.2 Message passing and delay tolerant networks

Communication in delay tolerant networks differs from normal networks since they require that messages are passed along, not only routed. The difference lies in the ability for a message to travel along time/space paths. This is done by passing on the message from node to node, it can be stored in each or some of the intermediate nodes if needed, before it gets passed further on. While routing, on the other hand, is merely a direct retransmission without storing.

By having the ability to store messages in intermediate nodes, on the way from sender to receiver, delay tolerant networks can reach not only the current network but also the network extended through the mobility of its nodes. An example to explain more easily; assume that two gas stations are located on the opposite sides of a broad high way, and the radio antennas can not reach one another. To get communication between the two, we could use the passing cars as intermediate nodes, by sending a message to a passing car, which later down the road passes by a car in the opposite direction, which it passes the message on to. The second car will later pass by the gas station, which now receives the message. This is one example on how to use mobility in an everyday scenario to gain connectivity. However, the same path used to send the message in one direction is not usable for the return messages. Each of the cars is long gone by this time. If no car would ever pass by the gas stations no message is returned. This asymmetry is part of the delay tolerant network's problems. Communication in one direction does not guarantee communication in the opposite direction.

The routing concept of message ferrying, as done by the two cars, is interesting in relation to information sharing. Both routing and information sharing are motivated by the same thing, routing, by getting the message to the one that needs it and information sharing, by getting information to the ones that need it. In delay tolerant networks the receiver's location is not known. The nodes in-between, the intermediate nodes, are also unknown. As the nodes move around, their encounters will lead to new network partitions and new space/-time paths, this behaviour has to be used to route packets. The protocol does so by placing the message in strategic positions so that the recipient receives the message eventually. The simple solution is to flood the message to all nodes in the partition and to every new node that comes in contact with nodes from the original partition. With this method we will surely reach every node by space/-time path that evolves. However, the nodes not reached by the flood technique can not be reached, since there is no space/time path to these nodes. This method of passing along packets is called, with some optimizations, gossiping or epidemic routing, the packet spreads like an epidemic or as gossip [25] [3]. The challenge is to restrain the gossiping so that the network does not get flooded with packets. The packet that is sent has to be self-contained, so that it can go out on the network and reach its recipient without the sender's aid, as the sender could be forever out of reach. This message passing will therefore not be able to deliver quality of service as a normal network, but has to be a "best effort" service.

2.1.3 Wifi limitations

When using MANETs it is important to be aware of the limitations of the wifi IEEE 802.11 [11] standard which makes up the connections. This standard runs in the 5 Ghz and 2.4 Ghz bands, and uses different types of modulation techniques. The most commonly used is 802.11g. This standard has a maximum bandwidth of 54 Mbit/s and is suffering from collisions with other standards and appliances like the 802.11b, Bluetooth, cordless phones and microwave ovens. This becomes an even bigger problem when there are many nodes running in the same sending area. Since wifi shares the same restrictions as any other radio technology that sends and receives on the same frequency, it is physically required to operate in simplex, only one at a time. This implies a strict control of who is sending and how the medium is used. Medium is here used to identify the frequency that the data is travelling on. The coordination technique is referred to as “Collision avoidance” which tries to make the medium collision free.

This mechanism is implemented with coordination messages such as “Request-to-send” (RTS) and “Clear-to-send” (CTS), in addition no packets, including RTS and CTS, will be sent before the medium is sensed to be unused [11]. When a state arises that calls for waiting or retransmit due to collision or corrupt packets, the standard calls for a random wait period, called the backoff period. This implies that in a dense network with many nodes and much traffic, the nodes are going to wait longer before sending, and colliding transmissions will becoming frequent as the network density grows. For two messages to collide, both messages will require to be sent at the same time, in this way is both nodes sensing that the medium is free. While both nodes are sending the packets collide in mid air, this however is not discovered during sending, both nodes need to finish their sending, and retransmission is called for, with all its overhead. This in turn makes for inefficient transmission and the network is then going to be rendered unusable due to poor performance. It is therefore not likely that ad-hoc networks can grow to large networks, over 50 nodes. As long as the network uses radio communication the number of nodes in one radio range can not grow large. If they do, the network would strangle itself and reduce the number of active nodes, while some of the nodes in a “full” area are not going to get the time slot they need to send packets, thus disconnecting them from the network. We can therefore assume small partition sizes when working with ad-hoc networks. This will have consequences for the later discussed implementation of the synchronization and dissemination protocols which have to use as little as possible of the bandwidth.

When talking about characteristics in radio and wifi communication, one should not overlook the good things about the shared radio channel. This means that any message is heard by any node within reach. In unicast all nodes that are not the recipient, drop the message, but broadcast can be done in a more efficient way than in a traditional infrastructure and wired networks. This is related to the same fact that also creates the limitations. In fact, the 802.11 standard defines the broadcast address to be all stations actively connected to the same medium [11]. This means that a broadcast call will be transmitted one time only, and all that hear it, receive it. In contrast you would in infrastructure networks have to retransmit the packet to all outgoing lines. This benefit will be used later.

2.1.4 Flood

Flood is a basic concept when using any network. It is often done by sending broadcast packets, and instructing everyone that receives one of these, to rebroadcast them. This technique can be efficient for some purposes and is often used for network services like topology information or other network wide coordination tasks.

Flooding of packets can be used to reach as many nodes as possible, and by doing so, locate one specific node, establish a route, gather topology information, or send packets, for example. The problem with flooding is that it uses enormous amounts of bandwidth. The exponential growth of messages will lead to the degradation of throughput if no restrictions are imposed. To look at an example of sending messages with flooding; the basic implementation is to send a broadcast message and instruct all to rebroadcast if they are not the recipient. By using this technique, each of the nodes hearing the first message will send one new broadcast message, which will be heard by all within reach, including the original node, which will rebroadcast the message for a second time. Every time a node which is not the recipient hears the broadcast message, it will make another copy. When looking at the end state, we will see that the recipient must receive all copies, while this is the only way to terminate a message copy.

There are many ways to do flooding with solving many of the problems included in this basic example [8]. The simplest techniques include; tracking messages so that each node never rebroadcasts a message more than once, others stop the messages after a defined number of hops, also referred to as time to live (TTL). By using such techniques together with broadcast messages, flooding can serve as a good technique to solve many network tasks.

2.1.5 The broadcast storm problem

When using flood and broadcast messages in the wrong way, serious network problems can emerge. This section summerizes the conclution from the article [16] with respect to usage in information shareing. Techniques and principles are proposed to enhance the use of broadcast and rebroadcast so that it can be used for efficient dissemination of messages in an MANET. Some techniques are listed in [16]. These techniques are in place to limit the rebroadcast of messages, where the rebroadcast does not add to the coverage of the dissemination.

The problem occurs when a node is trying to disseminate a message in the network using rebroadcasts from all the other nodes. As the rebroadcasts are transmitted, problems occur due to three reasons;

- Redundant broadcasts, the occurrence of two or more nodes rebroadcasting a message without reaching more nodes.
- Contention, happens when multiple nodes rebroadcast a message at the same time.
- Collision, while the IEEE 802.11 standard does not require broadcast messages to use the request to send and clear to send mechanisms, which gives broadcast messages a higher probability of colliding.

All categories of problems are caused by the overlapping radio transmission range of each node. A rebroadcast in a randomly generated topology, where all nodes have one common neighbor, shows that the average additional coverage by each node is as low as 41% [16].

The goal of the techniques proposed in the article is to limit the number of rebroadcasts, as many of them are redundant, thus avoiding the three problem areas. This is done through five approaches;

- A probabilistic scheme, where a probability P is introduced when rebroadcasting, so only some of the heard messages are rebroadcasted. To also handle contention and collision, a randomly selected back of period is used to avoid simultaneous rebroadcasts.

- The Counter-based scheme, where a counter is used to only retransmit messages that are not heard a given number of times. This is based on the fact that after hearing a message k times, the expected additional coverage by this node is less likely.
- Distance-based scheme, which uses the distance between the sender and receiver, if they are located at approximately the same location, their reachable area are the same, hence only one should rebroadcast. It is proposed to use signal strength to estimate the distance between the nodes.
- Location-based scheme, that uses GPS positions to greatly enhance the topology information, to establish an overview of the uniquely reachable area for each node, which is the area reached by only one node.
- Cluster-based scheme, which uses the transmission range of one node and places all its reachable nodes as its members, and the node itself as the head. Clusters also include gateways which can communicate with another cluster. Only gateways and heads are allowed to rebroadcast, hence reducing the number of possible rebroadcasting nodes, which can be further reduced by putting any of the above techniques on top.

From the results we learn that the location-based scheme is the most efficient, using the exact information of node locations and topology. The probability based approach has been shown to perform well even with a small probability, which needs to be increased for use in sparse topologies. As a side effect of the increased probability, will the savings of bandwidth is decreased proportionally to the increased probability, since more rebroadcasts will occur. The counter-based technique can obtain a reduction of messages up to $\frac{2}{3}$ depending on the number of neighbors each node has, if the counter threshold is set to 3. The distance based scheme is performing better in respect to reachability, since the correct messages are not rebroadcasted, however the savings is as large as the counter based technique. Unfortunately it introduces higher latency.

2.2 Message piggybacking

The technique of piggybacking is simple, the concept utilizes a message that is going from A to B, locates another message that also is going from A to B and appends the second message into the same network packet as the first. In this way the two packets travel together over the communication channel. At the receiving end the two packets are dismantled and treated as two packets. The piggyback technique can be applied to all layers of the communication stack, both lower levels and middleware can use the piggybacking technique. It is especially efficient in networks that always send packets of a certain size, and where the packet payload sometimes is less than that packet size. In these situations the gain of piggy backing, thus filling up each network packet, will be large. The use of efficient piggybacking is closely related to the nature and behaviour of the physical network.

In wifi networks the concept of piggybacking can be used to gain throughput between two nodes. The shorter each packet is the more packets there will be in the network. Each packet introduces an overhead in using the request-to-send and clear-to-send technique. This overhead is per packet, thus reducing the number of packets will reduce the overhead. Additionally, long packets will be less exposed to collisions, while collisions occur often due to simultaneous transmissions, so if the packet does not get any collisions at the start of the transmission it is not likely to get any collisions at all, and large quantities of data can be transferred without high collision rate. However, as packets are transferred, they are exposed to both collisions and noise, but this will not be identified until the transmission is finished, hence a long packet will send the entire packet before the data corruption is detected. This gives long packets a drawback while they hold up the transmission medium for a longer time, and if corrupted, the entire packet must be retransmitted, with the same problems as the first transfer.

2.3 Dissemination processes

Dissemination is used to describe the actions done by the network as a whole. The actions by each node contribute to the global dissemination. In this thesis we disseminate information in the form of metadata. This is done with the vision of having each and every node possessing one copy of each metadata element. To achieve it we need to disseminate the metadata from one node to all other nodes. The processes used to do so will be described in this section.

2.3.1 Publish subscriber

Publish-subscriber systems are an asynchronous form of communication which decouples the sender and receiving side of the communication. This is done by classification of each message, so that the sender referred to as a publisher is sending message into the system, not directly to a receiver. The receivers are subscribing to all messages of some class. The system routes messages from the publisher to all interested subscribers.

These systems are benefiting from low network load, by using multicast possibilities, not unicast from the publisher which is an alternative. Filtering of messages can be distributed, and as near the publisher as possible. It also has the benefit of the publisher not needing to maintain the list of receivers. This makes publish-subscriber systems very scaleable and well suited for MANETS.

2.3.2 Expanding ring search behaviour

Expanding ring search is often used when describing flood-controlling behaviour in MANETs. The concept uses the physical topology to limiting the search. This is done by limiting the number of hops a message can do before it is dropped. The technique can be used to search for a specific node, or for a node with specific information. In its basic form it includes sending a search query which is limited to only travelling a given number of hops. By expanding the number of hops, we limit the initial search ring to a small number of nodes and thus affecting only a small portion of the network.

By expanding the search ring from a small number of hops to a larger number the network load is reduced. To do this, one sends a message to all neighbors, for example, 1 hop radius. Then, if the recipient is not found the ring or number of hops is expanded to more hops. This is repeated until the recipient is found or a max ring size is reached. After this limit is reached, the ring search often uses alternative solutions like flooding.

There have been done studies into optimal max ring sizes [9] [10]. The tradeoff is between the repeated initial search rings, and repeated search of these nodes. The initial rings get searched every time another ring search is performed, while all searches start from one node. This gives a penalty both in network load and time consumed, on the other hand, if the recipient is found inside the first rings the rest of the network will stay unaffected, and the saving of bandwidth is large. When the full out flood has started the entire network must get the message without regards to the fact that the recipient already has received the message. Both mathematical and empirical studies show that the optimum ring size is locatable. It varies from topology to topology, based on size and density. Some have concluded with the use of a static ring size for nodes that have no knowledge of their topology. The tests and mathematics are based on random scenarios where both the topology layout and the placement of the searching node and the recipient are done randomly.

2.4 Classes of routing protocols

Routing protocols in MANET have been and probably are, one of the largest research areas inside MANETs. Many protocols and concepts for routing in MANETs have been developed. I want to look into if and how these concepts can be utilized for information sharing purposes. This will be handled in Section 4.6. The routing protocols for MANETs in general deliver two important services;

- Establishing routes for transferring packets to a destination.
- Topology information which is information about how the nodes are connected.

I will now go through the classes of routing protocols in MANETs, and go into more detail in some implementations showing good examples of the routing class.

We will use both dens and sparse MANET routing protocols. The difference in the protocols is based on which type of topology they can handle. We separate the dense MANETs, which are those topologies where every node reaches all other nodes in the network by a sparse path, thus there exists only one partition where all nodes are located. I will use the term sparse MANET when describing an ad-hoc network that also includes partitions. When this is the case, we have to use delay tolerant network protocols to route packets over space/time paths, while not all nodes can be reached by a space path.

2.4.1 Space path protocols

In this class we find the protocols that handle direct communication and alter their routes upon topology changes. They react to changes in connectivity and try to find the quickest route to other nodes in the network. Two main groups in this class are reactive and proactive [6]. A good and widely used example of the first is Ad-hoc On-Demand Distance Vector Routing(AODV). The protocol is not actively making routes until there is a need for such,either on demand or reactively. Then it starts route discovery and establishes a route to the destination. Reactiveness leads to less overhead in most cases. The protocol suffers from its lack of topology awareness. It is also dependent on stable routes and long communication sessions, on the other hand the overhead is only for creating and repairing each route. Since the overhead is associated with each new route AODV will generate more overhead as the sessions get smaller and new routes get more frequent. The group of proactive protocols is represented by Open Link State Routing (OLSR) which is, as the name implies, a link state routing protocol. It is proactive and constantly probes the network for neighborhood changes, hence keeping track of the total topology. It also eliminates the route discovery delay and overhead that reactive protocols can experience, but introduces a constant overhead that is used to monitor the topology.

I will now in greater detail describe the two major space path protocols. The two protocols exemplify the two subclasses of how to discover nodes and paths to nodes in a MANET. It is therefore interesting to look closely at these mechanisms.

AODV

Ad hoc On-Demand Distance Vector (AODV) Routing is a reactive protocol [18]. The benefit for the MANET when using a reactive protocol is that no traffic is generated unless there is a need for a route. This need can arise for two reasons. One, there is no route to the packet's destination. Two, there is a broken route for an actual packet. In both situations there is a need for a new route, but no information on where the destination node is located. The sequence of events that AODV goes through is described below;

- Send a packet called Route Request (RREQ). It is sent by broadcast to the network neighbors. An expanding ring search (described in section 2.3.2) is done to prevent flooding the entire network.

- If the correct recipient node receives the RREQ, it sends a Route Reply (RREP) back to the requester. The RREP follows the same route as the <u>first</u> received RREQ as this has proven to be the fastest route. Later RREQs received by the recipient node, with the same sequence number are discarded as duplicates.

- The requester receives the RREP and now the route is established. Intermediate routers that receive a RREP, find the first node that it received the origin RREQ from, update the route accordingly and forwards the RREP to this node, so that a message follows the fastest RREQ.

Routes established by this method, are active for a given time period starting from the last sent packet over the route. The established route is used for all packets in the consecutive transmission.

OLSR

Open link state routing [2] is based on link state routing from wired networks, which operates with a total view of the topology, also called the link state. This total view is obtained by periodically broadcasting HELLO messages which are received by neighbors. A HELLO that is received is indicating that the sender is a neighbor. If two consecutive HELLOs are heard, the sender is accepted as a neighbor and is so until two consecutive HELLOs are missed. It is these HELLOs that are making the major overhead of OLSR. The HELLO packets give each node information about its neighbors, which is combined with other nodes neighborhood information to form the global topology. The topology contains how many hops there are to any node in the partition, following the fastest route. The information from each node is entered into a packet which is distributed to form the global topology. To speed up this distribution process OLSR uses an overlay network which is dynamically created. Each of the overlay nodes gathers its "local" topology and distributes this to the other overlay network nodes, which propagate it to their nearby nodes. This happens each time new links arise or die out. The overlay network will make the dissemination of neighborhood information more efficient than simple epidemic dissemination or flooding.

OLSR Deamon, OLSRD is the major implementation of OLSR. It maintains the operating system routing tables so that existing transport layer implementations can be directly used in MANETS.

2.4.2 Space/Time path protocols

In sparse networks the probability of partitioning will be high, and the neighbor per node are few. Those nodes that are connected will form partitions. In these situations there can exist space/time paths between partitions. These paths are made up of each node's connectivity over time, as explained by using the ferry example: There is no road - space path - to an island, so to get to the island one needs to take a ferry. The ferry can represent a node that moves from one partition to another, like the ferry from the mainland to the island. Hence there is a connection between the mainland and the island, the two partitions.

This ferry technique solves some of the situations that space paths cannot. But we are still dependent on the movement from A to B and hence the connectivity with first A then later B. In addition we are also dependent on the intermediate node, in this case the ferry, to carry the message from sender to receiver and pass it along. We will see that for a more general case there is no easy solution to know how to "pass along" so that not everyone gets the message, hence flooding. We still need to ensure that the message gets to its destination and to as small as possible portion of other nodes. If a message is on the wrong ferry going in the wrong direction, this message will survive until someone deletes it. As we can imagine there are plenty of problems to handle for the space/time routing protocols. The selection of intermediate routes, techniques for limiting the travel range of each packet and intermediate nodes storage capacity are only some of the major concerns for the protocols. I will now list some of the protocols developed. This is only some of the protocols that exist but the collection shows the important techniques used to efficiently route packets in dens MANETs.

Message ferrying

Message ferrying is one of the main concepts in routing over time/space paths. But the message ferry (MF) protocol [27] that is presented, uses a more controlled environment than the MANETs in our application domain. The solution is based around an MF that is assigned to do the ferrying much like the island ferry. This MF has long range broadcast capability and a planned trajectory. This enables the nodes in the vicinity of this trajectory to prepare messages for transmitting to the MF. Transmission is done when the MF and the node meet. This protocol is more suitable for stationary or semi-stationary nodes and often there is infrastructure to support the MF.

Epidemic routing

The concept of epidemic routing is to disseminate messages much like a disease. It uses a protocol that makes messages spread out like an epidemic and after some time the whole network is infected, or enlightened in this case. This protocol [25] is a controlled flooding or dissemination of the packages. The protocol takes every message that is to be disseminated and adds it to its message buffer. When a node meets another, the message buffer hashes are exchanged and examined, and the complementary messages are transferred. After this meeting both nodes are equal with respect to the content of the message buffers. The message gets spread out through the network in this manner until consequently the message reaches its destination or the TTL counter is zero and the message is dropped. This protocol delivers the messages to the receiver if the intermediate nodes have large enough message buffer space to carry all messages requested of them. The drawback is that many of the nodes that receive a message were not supposed to have it. The overhead for wrongly directed or misguided messages fills up the network and consumes both buffer space and network capacity.

Probabilistic routing

To use the good concepts of Epidemic Routing and try to cope with the ever so full message buffers the probabilistic routing protocol gives an answer, but at a cost. The PROPHET [12] protocol makes use of which nodes a certain node is likely to encounter. This is done by maintaining histories of seen nodes. If B is encountered frequently by A, A is a good path to B. By using this simple history, messages are retransmitted to intermediate nodes if the encountered node has a higher probability to encounter the receiver than the node itself. The messages are not deleted since the sender node can encounter a better path at a later point. The effect of this protocol over epidemic routing, is that not every node gets the message, only the once that have a probability of meeting the receiver. This reduces the overhead, at the cost of maintaining history.

Location aided routing

Location aided routing (LAR) [26] optimizes route discovery from the basic implementation of flooding. This is done with the use of GPS positions by giving each node the ability to know its own position, and distribute this in the network. The knowledge of positions gives the ability to establish an “expected zone”, which is the area where the target node is probably located, constructed at the source from the last known position and speed vector. By knowing where the node is expected to be located, one can perform a more directed flooding to establish a route. By directing the flood of route discovery messages parts of the network will be cut off from the flooding messages, thus saving bandwidth. This can safely be done while the nodes cut off are not contributing to the discovery as they are not in-between the sender and the target node.

The cutoff is implemented in different ways. The simplest optimalization of flooding is done through establishing a request zone. The construction of the request zone is discussed, but is required to span from the sender to the target node. By only letting the nodes inside the request zone forward route discovery messages, the flood will be contained inside the request zone. Another approach is done through calculating the distance from each node to the target, and only permitting the nodes that have a distance less than the previous sender of the route discovery message plus some constant. This gives the route discovery message an ever-closing path, which will end up at the target. Results from simulations using the two schemes, show significantly lower routing overhead than by using the flooding scheme.

Semantic routing

In the application domain of emergency and rescue we have well-defined reporting routines. These are often along some hierarchical structure, where a fireman reports to his or her team leader, who reports further. These reporting chains are based on roles amongst rescue workers. By using the semantic role of each node, we can establish groups of nodes that either have the same role or reports to each other. By giving the groups first priority on new information, the reporting chains will always get new information fast.

When new information arrives at a new level in the reporting hierarchy the nodes around this group member will quickly learn about new information, while each node will go back to disseminating information the normal way after finishing with its own group.

Groups can also be looked at as overlay networks, where the group members are parts of the network. This scheme can also be seen in the emergency scenario where an information element meant for all team leaders quickly disseminates to all with this role, and thereafter disseminates to all team members. This is currently implemented via verbal messages, but the feature is wanted also for digital information sharing.

I have not found any implementations using semantic routing. Overlay networks and the existing reporting routines is the nearest implementation of such.

2.5 MIDAS

The MIDAS project is an acronym for Middleware Platform for Developing and Deploying Advanced Mobile Services. It has set the goal of making a middleware that will fulfill this vision: “simplify and speed up the task of developing and deploying mobile applications and services.” [15] The project consists of partners from all of Europe, and is funded by the EU. This thesis is concerned with the development of Global metadata manager (GMDM) for the MIDAS data space

(MDS). The GMDM does the job of data dictionary for the “data space” or database which MDS implements.

MIDAS also contains different components like Communication and routing (CRT) which addresses routing and message transport. CRT is in place for MIDAS to be able to use a variety of network technologys like; wifi both ad-hoc networks and infrastructure, and GPRS. CRT aims to handle transitions from AD-HOC to infrastructure networks like GPRS transparently. CRT is the only component that MDS is dependent on in the MIDAS context.

Other components handle context based routing these are not handled in this thesis.

2.5.1 MIDAS Data Space

MDS, MIDAS data space, is a middleware providing a distributed data space. The layout of MDS subcomponents and there relation with CRT can be viewed in Figure 2.1. We refer to the data space not as a database since it lacks many of the features that a database has. The largest differences are in transaction handling and locking, none of these will be addressed in MDS. Further the data space will not be able to guarantee total consistency due to possible partitioning. The data space resembles a relational database; we use a reduced SQL language as an interface. This reduction is described in the MIDAS documentation and is not the focus of this thesis.

MDS is designed to provide a common dataspace for all MIDAS nodes, where applications can share data. This is going to be done in such a way that the application on one node can enter data into the dataspace and other nodes can get the data out of the dataspace. The underlying techology to share the data is transparent to the application developer. The sharing is done with a combination of replication and remote queries. Further MDS will support data consistency amongst the nodes, so that if one node inserts more data into a table, this additional data should be available to all nodes within a short delay. [14]. A table in MIDAS terms is predefine before the scenario starts, and materialized once any node does an replication of that table. The table is there for existing as zero or more replications. However, a zero alocated table is per definition not dissiminated by the GMDM as metadata.

Global Metadata Manager

This thesis will in detail look into the inner workings and protocols of GMDM. The GMDM will serve as a data dictionary in the MDS. The GMDM will be fully distributed and handle merges and partitioning in a robust and dynamic manner. It is imperative for MDS to have a global view of the whole network and the information that resides inside. The GMDM at each node will try to get a view of the whole network with respect to metadata. This is done by gathering all the information and cooperation amongst neighbors. By exchanging information with every encountered node the view will expand. Information is exchanged with new nodes, which in turn will exchange with new nodes, thus disseminating through the network. The data dictionary implemented in MIDAS will be mainly concerned with the presence of relational database-style tables in the network.

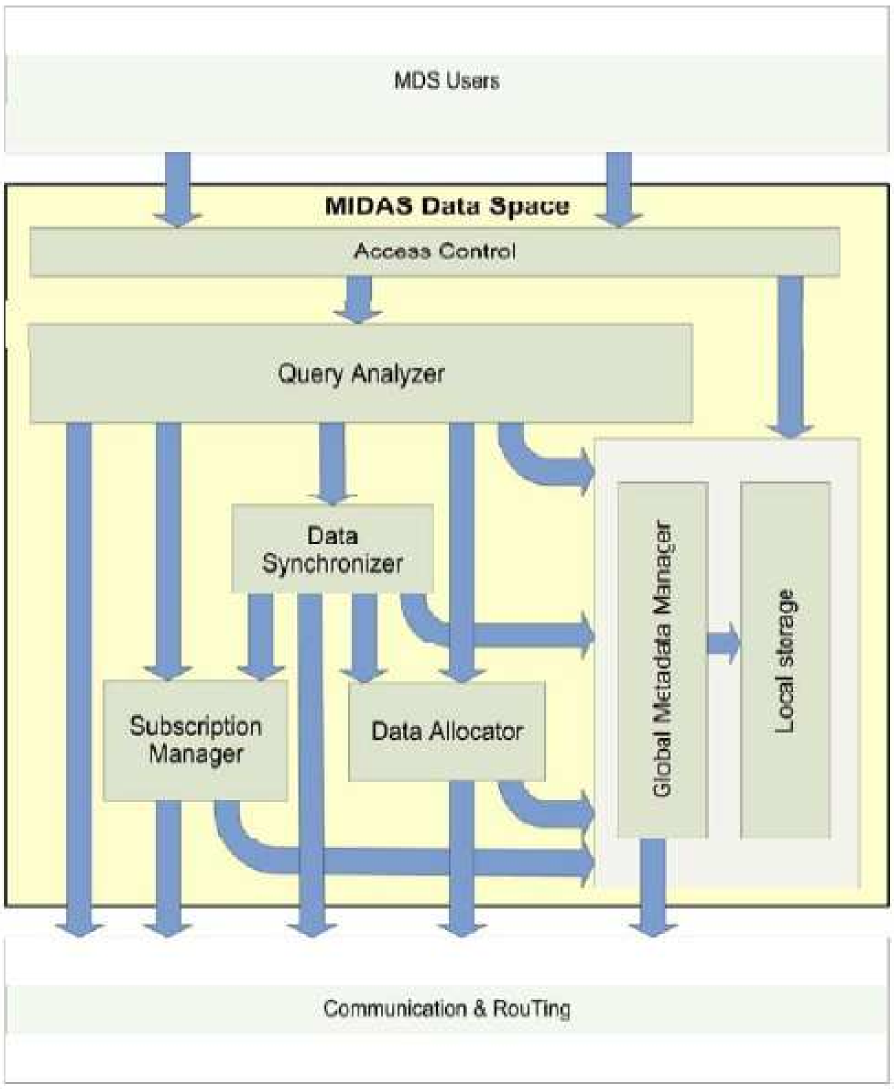


Figure 2.1: Showing the relation between the subcomponents of MDS and the component CRT. [14]

The GMDM will provide functions to other components that can give the node name where a spesific table replica is located. To do so the component will maintain internal structures of the discovered metadata. Maintaining these through eager synchronization. The overall goal for the GMDM is to use as little as possible bandwidth but still provide the resolution service.

Query analyzer

The Query Analyzer(QA) acts as a facade of MDS. It receives and handles any query invoked on the MDS instance. QA components on diffrent nodes can cooperate by sending remote querys.

QA uses other MDS subcomponents to find the node which has a table instance. If the table is residing on the local node, QA will use the local store (LS) to fetch the data. If remote nodes has the table instance, remote query will be used. This is done by sending a query as a message to the node, and getting a result back, which contains the result. These operations can be bandwidth consuming.

To handle the queries QA have implementations of advanced query analysis which identifies field names and values. From the field names and values, will the QA component identify which node that have the table instance. Some queries optimization is also done before remote queries are performed.

Data synchronizer

The data synchronizer (DS) has the responsibility of keeping all table replicas synchronized. To preserve the consistency DS will listen for both topology changes and queries. Upon any topology changes reported by CRT, the GMDM will notify DS it has discovered any new tables or table replicas. In this case DS will query GMDM for any table that is replicated on the local node and the newly discovered node. By using a defined protocol DS will use the QA's remote query services to synchronize the two replicas. Upon any INSERT or UPDATE query, the QA will notify DS about the change, this triggers DS to find all replicas and send a remote query, INSERT or UPDATE, to these nodes.

2.6 Consistency

Traditional databases are a large research field, and are in viedly usage. They imbed many features which releate to transparency of data, I will here focus on the features that are most relevant. A distributed database is a database divided over two or more physical locations, each of which are databases in there own right, with or without users of there own [5]. These database instances join together into a unified distributed database. The distributed database presents a single view of all databases, thus making them transparent. By doing so, the distributed database system will take on the job of keeping all the databases consistent. This implies locking, rollback and other database-related services. These services are commonplace when dealing with distributed databases.

A data set that is retrieved from a distributed database system, can physically reside in many locations, even though the transparency hides the physical location. Data can be distributed amongst the locations in different ways. The distribution can be fully replicated, where every data item is copied to all locations, or it could be fully distributed, where all is splited and fragments shared amongst the nodes with a minimum of data “duplicated” to ensure correct joins of all the data, this is called a nonredundant allocation. In the first case the data is easy to access and response time is good for queries, due to the fact that data is near the user. The downside is insertion time and consistency penalties. In a full-fledged distributed database system all local systems have to be in sync, so that from any location the same query will get the identical results. To ensure that the ACID properties are preserved, the insert steps transform all local database systems from the current consistent state to another consistent state, which can be difficult and resource consuming. When replicating data will the availability go up together with the resource useage, sticking the balance between the wanted resource usage and the needed availability is important to get the correct behavior.

It is also important to note that to implement a distributed database system that will apply to all the consistency requirements, one needs to make an intermediate layer, or integration layer. This layer is between the user application and the different database systems. This layer is often acting as a centralized unit or controlled by a centralized controller. This controller makes the calls about transactions management and replication strategies. The global data dictionary is inside this layer. Its task is to help the query planner and optimizer with information about where data is located.

The controller is the distributed database management system (DDMS), this controller will enshure that the database operated so that it does not violate any of the garantees it provide. There is a set of garantees that is implisit when working with databases, such as; availability, consistency, transactions and transparency, I will later argue that these are not fesable in a ad-hoc network. However, will the SQL interface be used as a interface, and some of the same features as thouse one can expect to find in a DDMS, therefore will i go into detaijl on which features we plan to support and which we can not support in Section 4.3

2.7 Simulation and emulation platforms

To test and verify distributed applications repetable scenarios are needed. This enables developers to isolate there applications and verify that there behavior are correct. I will now introduce some platforms for simulation and emulation. Both platform types are used for the same purpose, but have different characteristics. The simulator is only simulating the environment and the application. The environment and application are made in an easy to implement and fast to develop, often script like, fashion, which is only runable on the simulator. In contrast the emulator emulates the environment and lets actual applications run inside. This gives the developer an opportunity to test there application in a controlled environment. The emulator separates the application from the environment, whereas the simulator combines the two in one application.

2.7.1 NS2

Ns2 is a network simulator [23] which aims to help network related research. It is in contrast to NEMAN (see Section 2.7.2) not an emulator. The differences will be explained in the NEMAN section. A simulator such as NS2 is used to simulate network topology and traffic.

NS2 gives users the ability to describe movement patterns of nodes. These movement patterns can be "played" inside the simulator. The movement will be simulated forming the topology of the network, which will be used to simulate connectivity between the nodes, so that communication appears to be using different protocols like TCP, UDP and others. The performance and other attributes can be measured during the simulation, in any part of the simulator.

The scenarios in the simulator are programmed from different perspectives.

- Topology, defines the connectivity.
- Movement, how the nodes are moved, a snapshot of the movement, or locations, gives the topology.
- Nodes, the behaviour of each node.
- End points or agents. These agents produce traffic on the network, in the form of side traffic or noise. The agents can also be instructed to replay internet scenarios captured from for example a real internet service provider (ISP).

Benefits of using a simulator is that it is a high-level platform that supports fast development and proof of concept simulations. This makes the development time smaller but usability is not present because all code must be written for the simulator and can not be reused on real hardware.

2.7.2 NEMAN

Neman is an emulator that emulates the communication layers of the network stack [13]. This is done through the linux operating system by using its network interfaces. The NEMAN emulator makes new virtual network interfaces and offers these to the user application. This makes the emulator transparent to the application, making emulator developed applications ready to run on real hardware instantaneously. This feature does only exist in emulators, in contrast to simulators where one has to rewrite much or all of the code. In this emulator each of the simulated nodes is bound through the linux operating system to one of the virtual interfaces, also called tap interfaces. Each tap is then acting as a real interface as far as the application sees things.

The interfaces that are made at one single emulation server are all connected through a topology manager called "topoman". Topoman is controlled by an external interface through UDP packets. To alter the topology, topoman will connect and disconnect the taps, like a switchboard. The given connectivity at one moment is the topology of the network at that moment. Neman is using the linux kernel to establish connectivity, so no packets are leaving the server but are only bouncing back up through the correct receiver tap. This gives NEMAN a throughput that is far greater then what one would expect in a real life wifi network, or any other network for that matter. It is also important to note that the emulator does not introduce network failure, like one can introduce in NS2.

Neman is controlled from an external GUI. The GUI is reading NS2-like scripts called NS2 scenario files, which describe startup connectivity and movement. The GUI sets up the topology and manages the connectivity throughout the emulation scenario. The GUI is also able to send control packets to the taps inside the server, which can be used to control the user applications above the middleware. We will come back to how we use this feature in later chapters.

Neman gives the application developer a very good platform in terms of flexibility, where the developer can make code that can be used directly on the real platform, in addition the developer can use the technology and language of choice.

2.8 Ad-Hoc InfoWare

MIDAS is developed as a continuation of another research project called Ad-Hoc InfoWare [20], the two projects are similar in their vision but differ in implementation and approach. Both work in ad-hoc environments using wifi, both handle sparse and dense dynamic topologies. MIDAS is reaching outside of the scope of Ad-Hoc InfoWare, which concentrates on only ad-hoc wireless environments and rescue scenarios. While MIDAS focuses on data storage and availability, Ad-Hoc InfoWare focuses on semantic relation between data residing on different nodes.

Ad-Hoc InfoWare introduces a three level data dictionary. The three levels handle the data stored, regarding semantical granularity. The conceptual layer forms an ontology mapping the concepts stored. The concepts are linked to the physical information or table via a semantical level in-between. This three level architecture makes up the metadata representing the physical information. Concepts and semantical data are linked together to create a unified understanding of the data which is physically stored on some node.

This approach makes it easy to find the correct data also regarding the semantical content. The dissemination of the metadata consists of merging the semantical layers.

2.8.1 Distributed Event Notification Service

DENS is an event notification system for mobile ad hoc networks [21]. It has a collection of protocols that makes up a publish-subscribe system. It uses an overlay network, which is a subset of the nodes in the network. The overlay keeps track of the subscriptions and the notifications that are sent. It makes the publish-subscribe network more resilient, which fulfils the goal of DENS, namely to be used in emergency and rescue applications.

The overlay network is one approach to resilience regarding transmission and dissemination. Its purpose is a more efficient transmission of information trough the network.

Chapter 3

Global metadata management

The GMDM is discussed in detail in this chapter, laying out the main tasks, dividing its functions into the presentation and dissemination components. Further in detail describing the information sharing and synchronization techniques used to realize the GMDM component.

The task of the global metadata manager (GMDM) is to organize all metadata in the global MDS dataspace. This implys gathering and organizing the meta-information created by different nodes in the network. GMDM is going to be used for various tasks by other MDS components, as discussed in Section 2.5.1.

Information gathering is the most important part of the GMDM component. To achieve global scope some sort of dissemination techniques will have to be developed. These techniques are governed by the requirements stated in Section 1.3. It also needs to balance the needs for dissemination up against the local storage space. GMDM also needs to be delay tolerant, which implies that store and forward techniques must be applied.

To cope with these requirements GMDM must track metadata originating at its own node in addition to gathering metadata from the network. By also providing its gathered knowledge to any new neighbor, the dissemination take on a transitive behaviour, which is implemented as store and forward in delay tolerant networks.

To disseminate metadata through the network efficiently I have looked into how principles of routing protocols for MANETs apply to GMDM. The goal is to improve the dissemination according to the performance requirements (see Section 1.3) by taking advantage of the work already done in this field.

3.1 Data space consistency in GMDM

The distributed MDS contains all the information from all nodes in the discovered network, meaning the metadata found by GMDM will reflect all discovered tables replicas in the network. The metadata space is a list of table replicas available, or has been available for the node to query data from. The metadata space is constructed from the gathered metadata.

All tables that are to be used in a scenario have to be predefined by the MDS_schema according to the MDS documentation [14], each table is assigned a unique name, a parameter describing if it is local or shared, and info about its attributes. The parameter local or shared defines if this table is going to be distributed and allowing remote querying or limited to a local scope. The local value is used for node specific data that will not be visible from outside the node. The name, is unique for all shared tables, this means that if two tables are called the same, they are from MDS point of view treated as two replicas of the same table. This has serious implications on synchronization and consistency of the table. An example of this is if two partitions contain the table "patient". The tables are created in separate portions and have existed without knowledge of each other. When the two partitions merge the table contents need to be synchronized according to the consistency rules of MDS. These limitations to table names are done by MIDAS at an early stage in planning. The reason for this is to keep things simple with respect to implementation. When using the table name as identification we can both deduct the data type and table definition. The restriction of predefined table scheme relieves the middleware of this task, since this is know a priori, and can be looked up in a data structure. However, this limitation is only by convention in GMDM. The metadata format is extendable to the extent that it can contain these data type and table definitions. By doing so would ad-hoc table creation could be supported by GMDM.

If there is only one network partition the tables in the schema are residing on one node, until a request for allocation of a table instance is issued. Before this request, all inserts and updates will affect the same physical table instance. After allocation all table replicas will be maintained and synchronized so they contain the same data. However, this is only be possible when the two nodes holding the two tables replicas are in communication range, space path or time/space path. If there has been a separation of the two nodes the tables replicas are maintained in the two network partitions and a synchronization is performed when the two network partitions are merged, since these two tables have the same unique name. Consistency at the data level is maintained by the DS component of the MDS and is not the focus for this paper.

The GMDM will operate on a different level of consistency. It will try to get all metadata, using eager techniques, to all nodes. Metadata is not required to be the same on two nodes in two different partitions until the two partitions are merged and a synchronization is performed. Such inconsistency is tolerated as long as the two partitions have not yet discovered each other. GMDM will never remove metadata automaticcaly, since tables can become available latter. This implies that the GMDM knowledge base is ever growing. This problem will not be the focus of this thesis. I do not expect it to become a problem in our scenarios since they are small and short-lived. In MIDAS scenarios are limited to one or two days which also would not present any problems, with respect to metadata quantities. However the running time of the scenario is not the major factor in determining the size of the local metadata storage, it is determind by the number of available, shared tables instances and the number of replications, which combined give the maximum size of the metadata storage. In addition the size of each metadata element is small, in the range of hundreds of bytes. When looking at the space problem on each device this will not be a problem for the test scenarios in neither MIDAS nor for this thesis.

3.2 Information sharing

Information sharing is often linked to the term “knowledge management”, which is used to describe the control of information amongst collaboration parties or machine useable information between such. The motivation to achieve good knowledge management is to have the information available at the right time and place. The people making the decisions will then perform better, and the company will succeed. I will focus on the information sharing concept as the process of disseminating information elements to all possible needing parties.

In MIDAS every node needs to be aware of all available information in the network. The availability is tracked by the GMDM component, which holds tuples of information elements consisting of;

1. the name of the information element
2. the location of the element

Such metadata elements are easy to disseminate in the network due to its small size. Only when a node knows about all the metadata elements representing all the information, the node will have a full view of what is available. The dissemination of information is crucial to the functionality of GMDM.

3.2.1 Information sharing purposes

The GMDM will be used for two purposes

1. Find a node that can has a table instance or replica
2. Give an overview of the information available in the network

To handle the first point, the GMDM could do the search in a lazy way. The “user” could ask for a table, and the GMDM could go out and search for the table. This would exclude tables that are in space/time paths from the node due to the possibly long search time. It would also stress the network for every search of information. If we can assume that there is magnitudes more selects (pull) than create (push), all the searching would fill up the available bandwidth. Following the absoulte and performance requirements (see section 1.3) will this approach not be used.

To handle the second point, the overview, the search process could be run to gather all the information in the network. One would have very long delay times on an overview query. It would also not be feasible to establish a sensible waiting period from sending query to stop-listen. This is due to the space/time paths that could take forever to return.

The solution that is chosen, is to push all the metadata elements and by this eager dissemination of metadata information , establish an environment where all nodes know of every element in the network. This approach makes the selects easy to process and the overview we will get for “free” because we already have all the metadata in local storage. This makes for effective use of the total view, and opens for new usage where information is available as the scenario progresses, and the knowledge base on each device is growing as the information is gathered by the node.

3.2.2 Synchronization

To do dissemination information needs to be transferred through the network, from node to node. Each node is required to act in a router-like fashion and pass on metadata to other nodes. This is done through synchronization between neighbors, which has the side effect that the nodes are in addition to routing metadata information, also listening and gathering metadata. Each synchronization step is therefore crucial for the effectiveness of the whole dissemination.

There exists many synchronization protocols, but questions around how many messages is the minimum to get a full synchronization or how to best make a representation so that one can detect differences, is not answered or well-defined. I will lay out a first approach below.

3.2.3 Mechanisms of synchronization

In its most basic form a synchronization is started to achieve one goal, to make two entities equal with respect to some quantity. For this to work the elements in the quantity need to be compared and the differences between two quantities established. Upon finishing will the two nodes have the same metadata elements.

The basic outline of a mechanism is the following three step protocol.

1. View all elements in both quantities
2. Find differences
3. Exchange needed elements

Step one establishes an overview of the joint quantity. Step two identifies the differences between the two quantities. Step three equals out the differences by copying the differences to the needed party.

When this is done on a local unit without communication cost or distribution problems, the synchronization looks trivial. When we use synchronization we will need to handle both communication cost and distribution. We will only have the total overview of one quantity, the differences must be found on one of the two nodes without copying the entire quantity. I will now discuss how to handle this.

Neighborhood scope

The simplest protocol works like follows, see Fig. 3.1 for a graphic illustration of the different steps; Node A makes a hash of its local knowledge base,

1. Node A sends the hash to node B
2. B sends back the elements that A needs (the compliment of A's hash) which B can provide, plus a list of the elements that B requires from A
3. A responds with these needed elements.

This simple synchronization protocol is used by the authors of the PROPHET protocol [12] and Epidemic Routing [25]. I will argue that this is a minimal solution and that it works well in situations where there are differences between

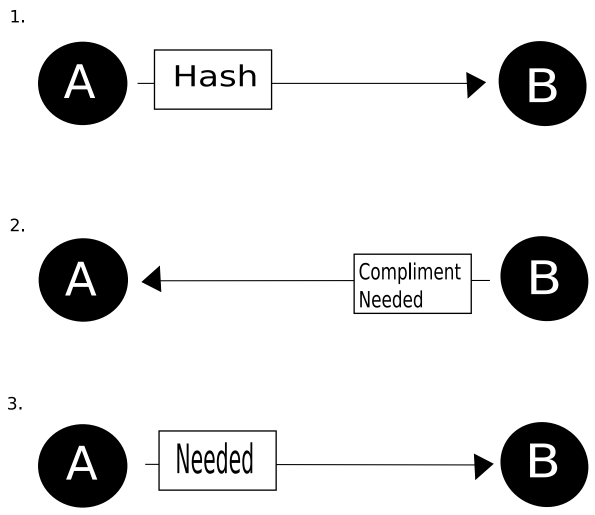


Figure 3.1: Simple sync protocol.

the nodes. In the case of multiple synchronizations between nodes that are "equal" this protocol could be optimized. I will look more deeply into this later, in Section 5.4.2. The GMDM will only rely on synchronization to disseminate information through the network. To guarantee the absolute requirements (see Section 1.3) namely that

1. all nodes that enter the network get all the metadata information of its neighbors

2. all new elements are disseminated through the whole network

To do so would we need to establish some triggers to start the synchronization. I will discuss this in more detail later, but for now I will state the triggers which we need for both 1 and 2 to be fulfilled. A synchronization has to start when everyone of the following occurs;

- a new 1-hop neighbor appears
- on the event of a new table created locally
- on the retrieval of new metadata

Metadata information entered into the network will be disseminated through the whole network if all three triggers are used. This will also apply to new nodes and nodes that are multihops from the nodes that created the new table. We will get back to this in Section 5.3.

Chapter 4

Requriments analysis and solutions

The Requirements analysis and solutions chapter aims to cover; the requirements MDS is posing on the routing implementation, and make decisions on which to chose. The MDS is placing some of its functionality in GMDM, these requirements together with the requirements placed by this thesis is discussed and defined. Indepth discussion of divergence between MDS and other standard distributed database systems, follow. More discussion into the physical nature of ad-hoc networks and wifi technology is done, with focus on who to use the physics to favor information sharing. Further is the mechanics of recursive versus multihop dissemination evaluated. Routing principles used for information sharing purposes are also discussed, with the intent of reusing the already developed techniques, an important aspect of the discussion is that of the counter effective differences, looking into where routing techniques are unusable for information sharing. The consept of overlaynetworks is discussed in relation to the DENS protocol. At the end of the chapter is an indepth discussion on how to implement the reduction techniques learned from broadcast storms, in information sharing systems.

4.1 Requirements put on GMDM

For MDS to function in MIDAS as proposed, there are some absolute requirements, some of these have to be fulfilled by the GMDM component. In addition to these requirements, some parameters will be measured to find the most optimal implementation, these are measured against the performance requirements. I will now go into more detailed specification of the problem description for this thesis, as first laid out in Section 1.3.

4.1.1 Absolute requirements

The following requirements are the foundation of the functionality in GMDM. Without fulfilling these, the implementation will be unable to play its role in MDS, and the functionality of other components will be affected.

- 100% dissemination. All nodes in the reachable network are required to know of all metadata elements. Metadata elements on nodes in one partition are required to disseminate to all nodes in that partition, and to all partitions that can be reached in a delay tolerant manner.
- Maintaining a global view. All nodes need to know where all table replicas reside. This requires that every node needs to store all metadata locally.

4.1.2 Performance requirements

The performance requirements state the optimal behaviour of the GMDM implementation. These reflect on measurements done under normal use and points out which parameters are important. No threshold is defined for any of the performance requirements, but the main goal is to have them combined as small as possible. Some of the requirements will probably affect others in a bad way. Optimization of one will decrease the performance of others, I will look into the balance between the requirements and try to find some optimal solutions. The balance is different in different scenarios and use cases. Use cases where there are more inserts than selects can gain much by balancing differently than use cases where the opposite occurs.

The performance requirements include;

- Low bandwidth. The dissemination should use as little bandwidth as possible.
- Dissemination speed. Time from the creation of information, until 100% coverage is obtained, should be as little as possible. Also dissemination via routes requiring delay tolerant techniques is also required to be efficient.
- Response time. Queries into information or metadata, like global view of tables and table name to node translation, should be fast or at best instantaneous.

If these requirements are balanced right, the protocols will be scalable. By scalable we require the implementation to handle an increase in number of nodes, density in the network and frequent trigger invocations. Cases where scalability will be visible and important is when partitions merge, since the protocols then need to handle all the new neighbors that appear and as the synchronizations progress, the new metadata that is discovered. Another scenario where scalability is important is if a application creates many new table replicas, this results in a dissemination wave, which uses bandwith as it progresses.

Balancing the implementation against longer response times, to save bandwidth, will result in less scalability for many new tables. This points out that the implementation is made to scale for its application domain, and should be measured against a scenario matching that domain.

4.2 Routing and topology service requirements from MDS

I will now map the requirements that one need is to be aware of when selecting routing protocol and topology services. When assessing what MDS demands from the routing daemon services, we have to assess the subcomponents GMDM, QA and SM.

GMDM requires as a minimum;

- the ability to broadcast messages to all one hop neighbors
- topology information in one hop radius

The ability to broadcast messages is needed by the GMDM to utilize the shared radio medium. Topology information is needed for GMDM to be neighbor hood aware, which is required if it is to react to neighbor hood changes. GMDM also needs global topology information and multihop message passing, to support more advanced synchronization protocols.

QA needs to;

- send multihop messages

This is a requirement for the remote queries to be realized in QA. In addition the QA is dependent on GMDM to provide information about which nodes host which table replicas.

SM is only using the QA component and is not adding to the routing daemon requirements.

Since topology information is a requirement, it is clear that a reactive protocol is not sufficient. Based on this a proactive protocol is selected, the most advanced and reliable we could find was the OLSR protocol; it maintains information about the global topology and can route packets over multihop routes, thus satisfying our demands. The routing is actually performed by the operating system, but is based on the routing tables on each node, maintained by OLSR. By selecting the OLSRD [17] implementation we limit ourselves to using OLSRD supported operation system on the nodes, currently this is available on both Linux and Windows in addition to other platforms. By using an open source third party project, like the OLSRD, we can focus on the main task, to develop MDS. The functionality required by the MDS subcomponents will be supported in the CRT component when it is completed by other project partners.

4.3 Dataspace vs. distributed databases

> We can define a distributed database (DDB) as a collection of multiple logically interrelated databases distributed over a computer network, and a distributed database management system (DDBMS) as a software system that manages a distributed database while making the distribution transparent to the user. [5]

This quote characterizes the distributed database and distributed database management systems. By analyzing using this quote and further specifications into what is required of a DDB and DDBMS, will I compare the requested behaviour of a DDB and DDBMS to what MDS is capable of delivering.

4.3.1 What differs MDS and DDB

A DDB is;

1. multiple logical databases
2. interrelation between the databases
3. communication over a computer network

MDS have one instance of a database on each node, represented by local storage (LS). MDS can, as a whole, be viewed as a DDB, since there can exist multiple instances of MDS in a network.

The databases can be interrelated, but only as replicas of tables is allowed. MDS will only replicate a table instance upon request and from then on try to maintain consistency between the two replicas. However, this is the only type of interrelation, since foreign keys and other relations between table instances on different nodes are not supported. A computer network is used of communication.

As we see all aspects are fulfilled, however, these similarities might lead us to the conclusion that MDS is a normal DDB, such a conclusion must be considered wrong. The characteristics of a DDB is both its behaviour but also its environment. Such a environment is not present for MDS, this leads us to the deviations from DDB to MDS. The normal environment where a DDB operates depends on stable connections. Stable in the sense that they are connected to the same router and has the same neighbors, thus the same databases are available over time. This enables coordination, which is essential for doing any type of transaction management or consistency.

The environment is not the same for a DDB and MDS. This shows that MDS can not deliver the same behaviour as the a DDB could. MDS handles this physical constraint and delivers a well defined service, described bellow.

4.3.2 What differs MDS and a DDBMS

A DDBMS does;

1. manages one or many distributed databases
2. making the distribution transparent

An DDBMS is the software that unifies the different instances of the databases. The main task of the DDBMS is to transparently make the all the instances visible as if there where only one [5]. This is done by adding a layer between the physical databases and the user, this layer is the DDBMS. The DDBMS software can use different database sources as long as the result is consistent and that it is presented transparently.

DDBMS is required to make many difference instances of a DDB transparent. The main issues that need to be hidden for the user by the DDBMS is;

- networks and location, the details of the network is hidden from the user. Location information or naming is not visible to the user.
- replication, data which is replicated onto different nodes are viewed as one single instance

- fragmentation, if data is shared amongst nodes, the view will defragment and combine the data with out the user knowing
- execution, how and where the queries and transactions are executed is hidden in the DDBMS layer.

Many of these transparency requirements, are not feasible unless one has stable connectivity, thus not in an ad-hoc network. The requirements are based on the nodes cooperating by combining the different database instances into a transparent larger database under the supervision of a central unit, or by a central unit. This implies that at least the controlling unit has to have connection with all the database instances. If this fundamental condition for cooperation is removed, no DDBMS could function.

The ad-hoc network can not give any guarantees to the connectivity. Any node can at any time disappear and never return. This underlying network nature stops any DDBMS to be able to grantee transparency, availability or reliability.

To strike a balance between the need for transparency, consistency, and availability, and the physical nature of the ad-hoc network, would one try to use the connectivity that is available and support a best effort service. This is done in MDS by giving a best effort dataspace. To handle the different types of transparency does MDS do the following actions.

4.3.3 Data space services

Network and location transparency are handled in different ways. Network transparency is not feasible to handle transparently in ad-hoc networks, we are not able to transparently cover over nodes that is shut-off or out of both space path and space/time path range. Location transparency can be handled, by giving the user an ability to query for tables names, and inside MDS transparently resolve the node name and its location, returning only the queried data, if the node is reachable.

Replication transparency, is semi handled. For a replica to be created would a user-command be issued, only then is the replica created on the specified node, which is a nontransparent replication. After the replica is created will MDS maintain consistency amongst the table replicas, as long as there is connectivity. When connectivity is lost, consistency is maintained with in each space/time partitions. This replication is transparent from the user perspective when querying for the data.

Fragmentation is not handled in MDS, it is explicitly taken out. This is done by not allowing queries to span over nodes. This means that a query can only contain one table that requires remote querying. Fragmentation transparency is therefore not an issue to MDS.

Execution transparency. The queries are parsed on the local node, identifying which nodes have the instance. The query is sent to this remote node, and the result returned, hence the query is executed on the remote node. Transactions, however is not supported, the closes we get is the inserts, if there are replicas and the instance are available, the insert will be forwarded and remotely inserted on each of the available instances. But the insert has no rollback function, and no exception handling, when replication instances are not available. If a table instance is not available will the instance simply be ignored, with out any effects for the insert. If the instance is later discovered, the table will be synchronized at the time of the merge.

As we can see from the handling of the different requirements, MDS does not satisfy all the requirements of a DDBMS. This is also the major reason for the naming choice, dataspace, while we will not invoke the association that MDS is, or will provide, DDBMS services. We will instead clearly state what we focus on accomplishing, and for these services only provide a best effort service.

4.4 Physical prerequisites

The shared medium which is part of the nature of wireless communication has some serious drawbacks. These drawbacks, when compared to normal wired networks forces all nodes to communicate in one single channel [11]. When all nodes share the same channel or medium it will easily be overloaded. As the load increases the network gets overloaded but it is not the only problem, end-to-end transmission delays and collisions become more frequent as the total capacity is filled up. The network gets congested when no one can get any messages through, the network is full of control packets. These control packets handle transmission coordination and cleaning up after collisions. These delays are emerging as the capacity of the channel or medium is reaching its maximum, and they will become larger as the number of nodes grows. Large delays are caused by the request to send mechanisms which coordinates transmissions and is part of the collision avoidance technique. In addition there are delays associated with any collision, since after a collision is detected each node will stop listening, and wait until the sender, which does send till completion, is finished, then the next transmission can start. While all share the same medium all transmissions must go in sequence, this requires a lot of coordination.

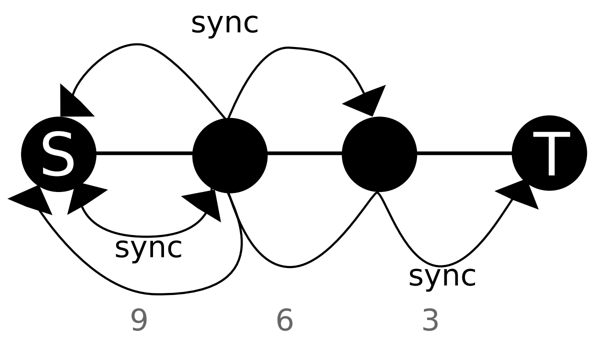
These prerequisites are part of the wireless nature, and we have to cope with them as laws of nature. However, there are ways to benefit from the physical shared medium. When all nodes need to use the same medium, this also means that all nodes share a common communication channel. This channel can be used by all to communicate with all, simultaneously. It is done through broadcast messages, which are addressed to all that hear them.

By turning disadvantage, the shared medium, to something useful, many to many communication, we are able to turn the drawbacks of shared medium to our advantage. The model of information sharing fits nicely with the use of the shared medium; since we want to share all information amongst all nodes in a network, which includes communicating from many to many. To be able to communicate and exchange information we require an efficient communication channel that supports communication. The shared medium and broadcast do exactly this. We will look into how exactly to utilize this advantage for information sharing in Section 6.2.2.

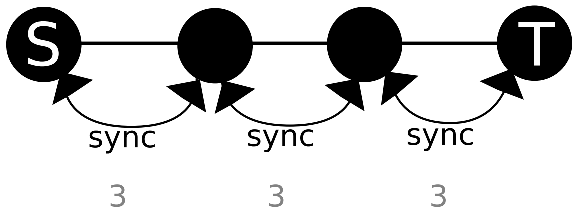
4.5 Penalty of multihop synchronization

When considering a topology and how to efficiently disseminate information from a source to all other nodes, one can imagine two schemes.

1. The source contacts all other nodes with the new information.



(a) Multihop synchronization between S and T, with the additional messages for 100% sharing



(b) Hop by hop synchronization between S and T

Figure 4.1: Multihop synchronization(a), compared with hop by hop dissemination(b). Each arrow with "sync" above is short for a 3 way synchronization protocol. Numbers in gray are the amount of network packets between each node during the entire scenario.

2. The source contact the neighboring nodes, without any intermediate nodes, and relies on a recursive dissemination by these nodes.

The first alternative uses the intermediate nodes as routers for the packets. By doing so will the source transfer its knowledge to one node, contributing nodes is only routing messages and not taking part in the information sharing. The intermediate nodes are also contacted individually, as recipients of the information in their own right. Since all nodes are required to get the information, is it more efficient to also share information with the intermediate nodes as it goes along, not only get them to route packets.

Consider the example in Figure 4.1, where 4 nodes form one partition. The nodes marked "S" and "T", source and target, are in this partition. In Figure 4.1(a) alternative 1 is illustrated , and the recursive method in Figure 4.1(b). The total network packets sent to disseminate information from S to all other nodes is 18 by using the direct method and 9 when using the recursive. The two different techniques have large differences when it comes to network packet use. To make the example more general I will now express the relation ship between number of nodes in a topology, and the number of packets used to disseminate information to all nodes in that chain. Assuming that all the parts of the synchronization protocol uses as much bandwidth with both approaches.

The multihop synchronization follows the formula in Figure 4.2(a) and the direct technique follows the 4.2(b). We can clearly see that the direct technique grows faster than the recursive synchronization. This shows that the differences

$$packets = (n-1) * 3$$

(a) Multihop synchronization, n is number of nodes

$$packets = \frac{n(n-1)}{2} * 3$$

(b) Recursive synchronization, n is number of nodes

Figure 4.2: Relationship between network packets and nodes. Requiring chain topology.

in efficiency between the two techniques exist, also in the general case..

4.6 Routing principles applied for information sharing

I will now study routing protocols for ad-hoc networks and see how the nature of routing resembles information sharing. One example is the epidemic routeing technique, which is made for delay tolerant and sparse ad-hoc networks. In this situation the sender does not need to be aware of the recipients location, or if it is in the network partition. The most efficient way, according to epidemic routing [25], is to disseminate the message to as many other nodes as possible, but still avoiding an all out flood. Epidemic routing is the relying on the nodes carrying the message to pass this on to new nodes when appropriate. By doing this, epidemic routing will strive to reach all nodes at least once.

This feature of disseminating in a controlled fashion is highly applicable in information sharing as well as in routing. They both share the same goal, to disseminate elements so that an undisclosed recipient is reached. By undisclosed I mean that in routing, we do not know where the recipient is located, and in information sharing, we want to reach all. The solution is the same for both problems. To guarantee that you will reach the recipient regardless of where it is located we will have to disseminate the message to all nodes in the network, which is exactly the same as information sharing is striving for.

Epidemic routing is part of the reactive routing protocols; which is the nature of delay tolerant routeing protocols. It will not be feasible to establish and maintain a global topology, because this would be required to include nonreachable and not yet discovered nodes. This implies that the reactive protocol needs to establish a route after the message is sent. This leads us into another interesting difference between the naturs of the two concepts. The reactive routing protocol is working to get the message to a specific node; the message is then delivered to the recipient and is no longer a concern for the routing protocol. Hence the message has a life time, in the eyes of the protocol, from sending to delivering. After this it will not be kept for later use. However in information sharing, the information or “message” persists from creation through sending, dissemination and until the scenario ends or the information is deleted. Information, as in information sharing, is valid and useful from delivery and onwards, where as the message, as in routing, is deleted and removed freeing space to other messages. While the routing protocols regard memory as a scarce resource, information sharing has to live with the fact that it will need the information over time, thus needing larger buffers and storage to keep the messages in.

When a protocol, information sharing or routing, receives more and more information or messages it stores these new elements in its local storage. If the storage is becoming full, the protocol has to remove elements to make room for this new element or drop the new element. In routing this will be acceptable and logical to remove the oldest element. In the testing of the Epidemic routing protocol [25], there is presented an relationship between the available storage space and arrival rate. As the storage space gets smaller, the arrival rate drops, because messages never reaches its destination before the message is pushed out of the buffers by other messages. This further implies that the local storage in information sharing protocols is to be large enough to hold all information gathered through the scenario. Deletion of random will violate the absolute requirements of 100% dissemination. While information will disappear from every node with a overfull local storage.

4.6.1 Counter effective differences

There are also some differences that are counter effective for information sharing, but are shown to be good for routing. While epidemic routing are based on the fact that we can not locate the recipient via the normal sparse ad-hoc network, there exists routing schemes that is base on the opposite. Location based protocols use the position of each node to aid the routing of messages. These protocols will try to find the direction that will most rapidly deliver the message to its receiver. This requires knowing the position of the receiver, but since this is known, it will be able to only send message in this direction, while epidemic routing had to send the packet in any direction to be certain of delivery. This implies using bandwidth also for those directions that does not contribute to the delivery, to the save messages used in the direction that does not contribute to delivery, one can use the direction of the recipient prior to sending. This aid will starve portions of the network if used in information sharing, because the message is constantly delivered to new nodes. It is effective when looking for one specific node and counter effective when addressing the whole network.

The “Location aided routing” (LAR) [26] is one of the location based routing protocols, most of which are based on link state routing, hence a presending known topology which is often found in dense networks. LAR is using a concept called, last known position, this concept uses the physical position of each node so that the routing protocol in addition to the connectivity also knows the position, and distance and movement can then be calculated. This principle is applicable to delay tolerant networks as well as dense networks. These protocols use the physical position to reduce the request zone, or transmission area. This has the effect that only the nodes in the request zone is receiving messages, thus counter effective to information sharing where all nodes, also those positioned outside the request zone, requires the information.

Another more basic technique to reduce the overhead of message routing, is to use a time to live (TTL) constraint. This constraint is in place to ensure that packets that does not reach their goal, stops after a given number of hops. This reduces the overhead of misguided packets going forever. This is once again a counter effective constraint in place to enhance routing by cutting off some portion of the network. The TTL scheme works in the following manner; the TTL counter is reduced by one at each node it visits. If the TTL counter is zero, the packet is dropped. This will in fact stop the message from reaching nodes that are TTL+1 hops away from the sender. In delay tolerant networks we do not know how many hops will be needed to reach all nodes, such a constraint will only limit the dissemination of information. And once again will the constraint not be suitable in it self for use in information sharing.

As the differences are laid out, there arises a separation of the mechanisms used. One, those that limits the number of recipients, which are TTL and location constrains, for example. Two, those that make dissemination to all more effective, and dynamic. Like epidemic routing and other store and forward mechanisms. I view this as a guide to which mechanisms to adopt and which not to use. The reason for this is that some unwanted side effects in routing are wanted effects in information sharing. This is in fact quite natural when we look at the overall goal of routing, which is sending from one to one, while in information sharing it is one to many. Despite these differences our intermediate goals are the same, use as little as possible of the available bandwidth, be fast, and deliver at least once.

4.6.2 Propabalistic routing used in information sharing

The concept of propabalistic routing is taken from the PROPHET protocol [12]. In this protocol the author tries to handle the overhead seen in epidemic routing. This is handled by building a statistics so that an educated guess can be made whether or not to forward a message to a encountered node. Once again is this constraint of not forwarding to everyone is made to limit the message to portions of the network. The author of PROPHET has shown that this gives less overhead and network usage but still performs at least as good as epidemic routing. This might be counter intuitive, since messages are replicated onto a subset of all nodes in the network hence giving a smaller dissemination rate. The reason why this works is that the nodes selected for dissemination are the nodes with the highest probability to encounter the requested node again later, based on statistics.

The protocol works by sending message from one to one. But the principle of using a dissemination with a probability is interesting also for information sharing. If we can deviate from the principle of 100% dissemination for this technique, it would be possible to see a scenario where we are able to disseminate to a large but not total portion of the network. If this is achieved in a uniform way, such that if all nodes as part of a partition could reach all information within a specific number of hops, would we be able to both hold the search time low in addition to lowering the dissemination cost. This scheme is feasable, if one could pick out a section, a neighborhood of 4-5 nodes that where connected via direct one hop links and these nodes combined had knowledge about of all information in the network.

At each node we would keep some portion of the global information base. Upon a request for a specific information element, would we first search the local storage, if nothing was found, would we do a limited ring search until we found the element. By doing this would we lose the instant search time, because we have to do a network search. The gain is in how the dissemination is done, when we no longer need to disseminate to all nodes. We could disseminate to a fraction of the neighboring nodes, the nodes had to be randomly picked so that the information would not be located on some nodes but evenly spread out over all the nodes. This dissemination would leave some of the nodes in each partition without some elements. The transfer that was not required to obtain 100% coverage, is the bandwidth gain of this technique. The larger the partition the less traffic overhead, but the larger the search time. To implement such a technique efficiently one has to find the governing trends into how the system is used, many searches will favour high dissemination rate, and many new creations would favour a low dissemination rate and a more time consuming query.

Two problems are still open when using this scheme;

1. the protocol must end, it needs a defined end state.
2. mechanisms to establish confidence that the distribution will become uniform.

The epidemic routing protocol stops when achieved 100% dissemination, defined by each and every node having all its neighbors equal with respect of the information storage. We can conduct that if each trigger is followed, all information will be disseminated to all. In the case of the propabilistic routing the deduction is harder, since not all triggers will result in a synchronization. If, by chance, no nodes start a synchronization with a newly arrived node, will the new node be starved.

To get the dissemination uniform we depend on the topology. If the network is too sparse we could be in a situation where some partitions of the network would never be reached. This happens, for example, if nodes that act as the single bridge between the partitions fail or decide not to forward their information to the other partition. If this happens, and no other node forwards the information, the entire partition will be starved for information. To handle these situations we need to detect that such a situation arises and respond to that situation so that no partition is starved. Both detection and handling is hard.

The PROPHET protocol takes advantage of that the dissemination protocols are redundant while they deliver the same information over multiple paths. This happens in networks where the nodes have a high connectivity which leads to many routes from the sender to any other node in the partition. By not using some of the routes we can still get the dissemination to a satisfactory level but with less overhead. These problems and questions need to be investigated and answered, these are outside the scope of this thesis and will therefore not be discussed further.

4.7 DENS message propagation model in information sharing

DENS combines routing of message and the knowledge of clusters to do dissemination of subscriptions, this saves bandwidth but introduces requirements into partitions and ability to negotiate hierarchies among the clusters.

DENS, distributed event notification service, uses a one-to-many propagation model to disseminate events through a subscription service. The subscription service is distributed in an overlay network spanning the network. DENS is transporting events in the form of messages from a publisher to all its subscribers, doing filtering as near to the publisher as possible. The dissemination process used in DENS is similar to the semantic synchronization protocol. What DENS does is building a hierarchy amongst the nodes. The top level nodes are part of the overlay network; each of these has some underlying nodes. If one of the underlying nodes holds a subscription to an event that is published, the message will follow the overlay network to the correct top level node and from there be forwarded to the correct sub level node. These sub structures of nodes forms "highways" for message transfer which can be stable or fast routes. Other motivations for building such structures are priorities for message delivery, where the nodes in the overlay structure have first priority and other nodes will be served after these. This is the case if all roles of some kind will require a fast dissemination.

I hope to use this overlay network to get higher dissemination speeds, with first priority amongst prioritized nodes, and hope to larger global dissemination. This can be done by using the overlay network and from each of the overlay network nodes out to the rest of the network. This gives the dissemination multiple starting points hence a parallel dissemination process. It is thought that this can give us faster dissemination speed than with basic epidemic dissemination.

4.8 The broadcast storm problem

The broadcast storm problem (Section 2.1.5) introduces some problems for routing and message passing in ad-hoc networks, these same problems are also occurring in information sharing. Measures to handle the problems are also described in the article covering this subject [16]. These measures are applicable to information sharing, as the dissemination can be viewed as a series of rebroadcasts, propagating information from node to node. As each new node gets new information, it triggers a new synchronization, which will engage all its neighbors. The problems occurring in broadcast storms are therefore also occurring in information sharing, we can also assume that the techniques used to enhance performance and avoid broadcast storms can be used to handle the same problems in information sharing.

As described by the broadcast storm article [16], we want to reduce the number of overlapping rebroadcasts or synchronizations. We need to reduce the number of synchronizations initialized, not only the number of messages sent.

Multiple techniques are presented, to reduce the broadcast storm problem, some of which are directly applicable to protocols used to solve the dissemination problem, others are good enhancements both types will be part of future work. I will now go through the different techniques;

Probabilistic scheme This technique is directly applicable to the protocols used for information sharing. One can use the probability to determine if the newly received information is going to trigger further synchronization. This will give less initiated protocols and thus less overlapping broadcasts. The downside seems to resemble that of the PROPHET protocol, which effects reachability. As described in Section 4.6.2, this can lead to starving of sparse connected partitions. To reduce the collision and contention, upon multiple initiations in the same coverage area, a backoff timer can be used, this measure can be used alone to se if the collision and contention rates will decrease .

Counter-based scheme Statistics of this behaviour can be used to determine whether or not the information is going to be disseminated by this node in the future. However, our situation is a bit different, since we can construct a message not only based on one other message but based on multiple sources, previous messages, local storage and the received message. To determine if the local storage is to be disseminated, based on the last received information element, can be misleading. But one can portray a solution where statistics are gathered on a per metadata element basis, and from these one can read how many times a metadata element is received over the last n seconds. These statistics can be used to build the dissemination packet, if it ends up empty, it will not be sent.

Distance-based scheme This scheme uses the distance to each node as a measurement onto how large contribution each node can make to the global coverage area by retransmitting. A small distance to a transmitting node means that the receiving node largely overlaps the transmission area for the sender which is already covered, hence little extra coverage leads to stop in the rebroadcasts. Overlapping disseminations can also occur when sharing information if nodes are transmitting in the same area, which is exactly the same problem as for broadcasts. If we were to measure the distance between the sender and the receiving node, we could use this information to evaluate if the receiver was contributing enough to the coverage area to be allowed to continue the dissemination. The distance measurement is hard, but can be accomplished by using the receiving strength or GPS positions.

Location-based scheme This scheme can use information from location aided routing protocols to calculate the accurate additional coverage by each node. It uses information from a GPS to evaluate the coverage area of each node, and from this information decides whether a node is allowed to continue a dissemination. In contrast to the distance based, where a approximation is done based on the signal strength. GPS information is not known to a MIDAS node as part of the project, which stops the implementation of such a scheme.

Cluster-based scheme The cluster based scheme uses a hierarchy of nodes, where there exists a cluster head, members and gateways. A cluster is formed by the members, a member is a node inside the range of the cluster head. The gateway is a node with connection to another cluster. The cluster is not a partition but a group of nodes inside a partition.

This hierarchy gives us a further optimization into which nodes can redessiminate information. This scheme is also applicable in information sharing, and will probably give good performance while the number of nodes generating packets and traffic is small, hence further reducing the bandwidth use.

4.8.1 How to use these schemes

Many of the schemes are applicable to information sharing. The broadcast protocol, which is the most advanced protocol in this thesis, suffer from many of these symptoms. As discussed before are NEMAN hiding some of the problems by not giving a realistic wifi environment. However, for real life usage would the broadcast storm techniques have to be implemented to avoid problems.

At first should the problem be analysed more thoroughly, by doing so would we locate the most pressing problem areas. At the moment not all schemes would be feasible to implement due to physical constraint like lack of GPS positioning, other is hard to implement in a general fashion since they involve cross layer communication, like he distance based scheme.

Both the probabilistic and counter based scheme is feasible to implement with out altering the node configuration. In addition would the cluster-based scheme might be feasible to implement by use of the already constructed OLSRD integration, since the entire topology is known. If these first implementations would give promising results, should one consider installing GPS or other positioning devices to look into the further gains of the distance and location-based schemes, since these are the once that seams most promising in the article results.

The implementation of these schemes introduced by the broadcast storm article is not included in this thesis. This is due to lack of time caused by the late discovery of the article.

Chapter 5

Design issues

This chapter defines the design choices done. By using the previous chapter combined with the requirements are choices made and described. First the GMDM architecture, then the metadata filtering position. To avoid information overflow will some component or the application filter the vast amount of information, where this filtering is done is discussed. The scope of propagation is discussed with relation to multi or one hop synchronization. Further are synchronization protocols discussed, and different approaches are layed out. The messages used in the protocols are also described.

5.1 GMDM architecture

GMDM is divided into two parts; the information gatherer and the information organizer. The gatherer has the responsibility of gathering and disseminating metadata to neighboring nodes. The organizer does the structuring of information to later be able to respond to queries from other nodes.

To maintain the data structure in the organizer up-to-date will it rely on the gathering part to fetch and disseminate metadata. As we can see will the gathering part update the organizer with new information. However new information about $l$ocal tables are updated directly in the organizer's data structure by the user application. This gives the two parts a tight coupling while they work on the same data structure.

5.2 Metadata filtering

The general concept of GMDM is to give a total view of the information in the network, which will be done in an eager fashion. That is, queries about table replicas are always placed to the local node. The knowledge in GMDM is used in two ways, either as a search for availability, or as a request to give an information overview. The search returns the name of a node which holds one replica of the table name in question. The overview request returns a collection of all table names whose replicas exist in the network.

When the application requests an overview, the middleware is providing a total view. In doing this the application is given all metadata information that is recorded by the GMDM. The information returned can contain both elements that are not reachable within this network partition nor needed by the application. The knowledge of which elements are needed by the application is not known to GMDM, thus it provides all information available to all applications. The main point is that filtering of which elements to return, is not up to GMDM. Applications can filter on the availability of the nodes or over other criteria. GMDM is providing the full set of metadata, giving the application the power to filter as it wishes.

It is important that the application is aware of this fact, since it has to handle the problems stated above. Application designers have to understand the benefits and problems this approach introduces.

It is important to note that this placement of the filtering introduces an overhead. This might be unavoidable, but the metadata information not used by the application has both used bandwidth and memory to get to this node. This part can be solved through better cross layer communication, where the application instructs the middleware about what information is valuable. In this way the middleware can filter out which information to request in a synchronization, and in this way save bandwidth. This approach is problematic in two ways; one, the saved bandwidth also stops dissemination. In some cases the needs of the middleware and the global dissemination must be set above the application. Two, as reasoning over information semantics gets better, information that is not thought of by the user can be found by algorithms. In this way information that is important to the user will not be hidden. This reasoning is dependent on knowing all information available. We have made the decision to filter only at the application level, to best avoid the problems encountered otherwise, as the subject these problems are outside the scope of this thesis.

5.3 Propagation scope GMDM

When discussing both dissemination and propagation we need to keep clear what the scope of the dissemination is for each node. We can look at the scope on two levels: at the global level and at the node level. At the global level the scope is all nodes in the network. We see the dissemination as a propagation from one to all by every one relaying messages, thus disseminating the entire network. At the node level the scope is limited to the nodes in one hop proximity, also referred to as the neighborhood. The node level scope is limited to these nodes under the argument that this is the most cost effective way, both with respect to the ever changing topology and the penalty of routing messages over intermediate nodes, as multihop synchronization. By keeping the scope to the neighbors the global relay functionality is still intact.

As the same rules apply to all nodes, we know that if a message is propagated to one neighbor, its neighbors will also get the information. We know this because all nodes apply to the same triggers (see Section 5.3). The dissemination will have a transitive property that ensures that the entire network will be disseminated.

Triggers

As we shall see there are three situations where the node needs to take actions to ensure that the transitivity holds. If one of the three situations is not handled correctly, some part of the node network can be starved or the dissemination can be delayed. The three triggers stated below is a complete set of rules to ensure transitivity

1. new metadata created
2. new metadata received
3. new neighbors

The rules govern which events trigger a synchronization. The triggers need to be constructed so that the individual nodes contribute to the global dissemination. This is done through analysis of 1) how metadata can be inserted into the network, covered by trigger 1, 2) how new metadata is discovered, covered by trigger 2 and 3. The goal is to use as little time from new information elements enter the network until they are fully disseminated. We also need to look at how new nodes are able to learn new information from networks they join, hence the expansion of the network with respect to new information.

Metadata can also be discovered through new nodes, therefore trigger number 3. Which ensures that metadata on new nodes arriving at the partition is discovered.

When we look at expansion of the network, we think both of the global information network, containing all table instances and the physical node network. The information network can be expanded by receiving new information from a neighbor. The node's local knowledge is then expanded, which makes it larger than the neighbors not in direct contact with the providing node. It is therefore up to this node to deliver the information to its neighbors, by providing the new information, as metadata, to its neighbors.

5.4 Synchronization protocols

The synchronization protocols are responsible for dissemination of information from one node to the next. The protocols are set into action by the GMDM after selecting two nodes that need to be synchronized.

To be able to describe the actions and procedures of the protocols I need to establish a difference between the listener and the initiator. This distinction is important when I later describe behaviour. Upon one of the triggers, a node will start a synchronization. The node that starts this synchronization is called the initiator. The protocol synchronization contains messages, the ones that receive these messages are called the listeners.

5.4.1 Messages

Any communication between nodes is done through sending network packets. These packets are the bits and bytes that traffic the network. To more easily use this communication, there is often made several abstractions. We use the java implementations of sockets, which abstract away the actual sending of the bytes, and CRT which abstract away the byte orientated packets, so we can send messages. A message is a defined object that contains; a sender, a reciver and a component name, thus uniquely identifies the receiver and the sender of each message. This basic message object can be extended to whatever usage the components will need.

Each synchronization protocol contains a set of rules, defining the message formats, and the message order. Rules define which messages are required to start a protocol, and which behaviour is required upon each message. The rules combined give a deterministic behaviour model on each of the involved nodes. For every event that can occur at any state, there will exist an action to perform. The actions performed for a protocol - to go from one state to another - is often combined of a trigger and an action. The trigger is most often a message received, the kind defines which state to go to, and which action to perform. However timeout triggers can also occur. Actions is typically constructing and sending of messages.

The protocol defines the communication between nodes as well as the behaviour on each of them. Clear definition of the messages is important for the protocol, since the message type and content decides the actions of the protocol.

To track the behaviour and status of the protocols each node will have some form of loging service. This services will provide any implementation with the nessasary functions to both mirror its status and its behaviour. The diffrence between the two and there usage is more thuroughly discussed in Section 6.7.

The messages are represented as a MIDASMessage, which is a java object. All messages used in the protocols are subclasses of this message. The message object that is going to be sent is serialized via the java ObjectOutputStream to from a byte sequence. At the receiver end the byte sequence can be reconstructed as a MIDASMessage of the correct type. This is not an optimal solution while the serialization is not as small as a handmade message type, but the message size has not been the main focus of this thesis. Therefore we have used this simple solution. Further work can study how large the savings are in relation to such a change in message layout. The message hierarchy is designed to do the message type transparent at the CRT component. The UML diagram in Figure 5.1 shows the message types, and the content of each message.

Overview message

The overview message is doubling as both a start-up message and an information hash. The semantic meaning of the startup message is that of starting a new protocol session. A listener that receives an overview message, will start up a new instance of the correct protocol and link the session to the protocol. More on sessions of protocols in the implementation, see Section 6.4.2. For now the overview message is only signalling that a new protocol is starting, and that this is the first message. The information hash is the primary use of the overview message. Inside the message there is a representation of the entire knowledge of the initiator node. The hash is made up of a list of unique representations for each metadata element, in MDS this is a tuple - table name and location - which is the MDS unique identifier of a metadata element.

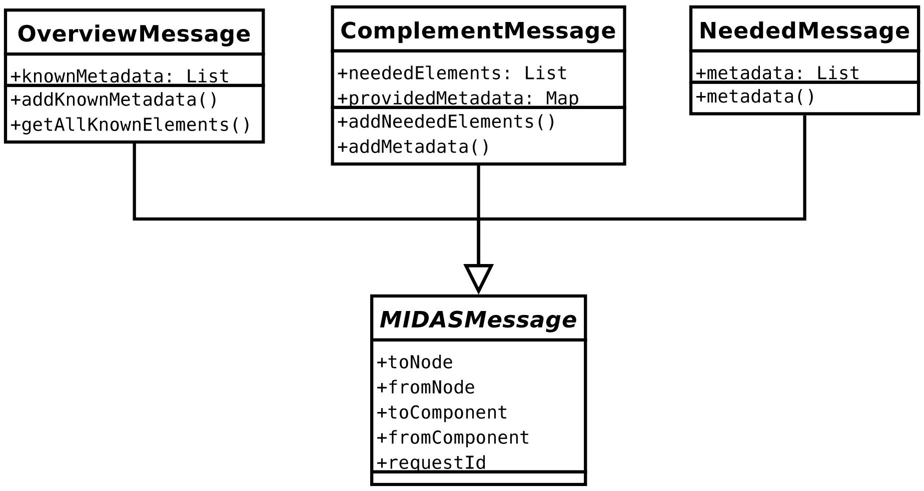


Figure 5.1: Shows the message objects

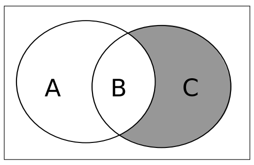


Figure 5.2: A Venndiagram representing A and B, the overview knowledge, C local knowledge, B shared knowledge

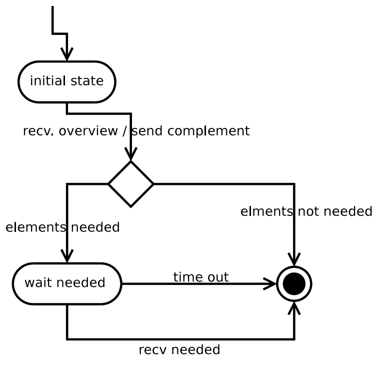
Complement message

The complement message is constructed as a response to the overview message. The information hash inside the overview message is assessed and evaluated against the knowledge locally. The elements that only exist in the local knowledge are entered into the complement message as shown in Fig 5.2 as the grey area, C. Note that we enter the entire metadata element into the complement message since we want to copy the element to the initiator. This means that metadata can have more attributes than inside the hash value. This is valuable for extending the metadata content.

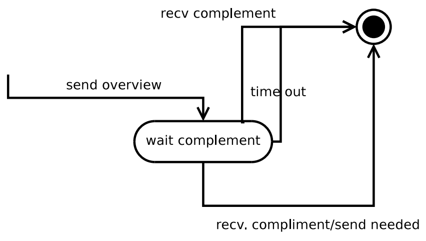
Along with the provided metadata elements in the complement, is attached a list of needed elements. These are the elements that are in A but not in B or C as shown in Fig 5.2. These are the elements needed by the listener. The list entries are the hash keys of the elements as represented in the overview message, and an entry is treated as a request for a given element.

Needed message

To be able to respond to the request for elements as provided by the complement message, the initiator uses the Needed message. This message transfers the elements added to its payload to the listener, it is more or less a simple envelope where metadata can be put.



(a) At the listening side



(b) At the initiating side

Figure 5.3: Epidemic propagation protocol represented by state machines.

5.4.2 Synchronization of already synchronized nodes

Each synchronization tries to even out differences between nodes. This is done after each of the described triggers, however there can occur situations where nodes that rediscover each other are already synchronized. This happens if two nodes are done synchronizing, loose contact, and later regain contact without either of them having new metadata elements. In these situations the nodes will start synchronization, but since there is no need for a synchronization all communication will be overhead. The implementation will decide how large this overhead will become, however some communication is required to identify the situation.

The protocols should have mechanisms for terminating this type of synchronization at an early stage. How this is done depends on the implementation, as the entire metadata hash representation has to be transferred over the network to the other node, it represents huge amounts of overhead traffic at worst.

The optimalization in the broadcast protocol tries to handle this problem to some extent. However, there are not done extensive tests and measurements into how large these problems are. This should be considered when doing future work.

5.5 Propagation issues

I will now describe the propagation techniques that will be implemented.

5.5.1 Epidemic propagation

Epidemic propagation uses concepts from epidemic routing. The protocol will try to pass every element to all new neighbors. This will result in the propagation of an element in a hop by hop fashion. Fig 5.3 illustrates the protocol as a state machine.

The protocol consists of two sub protocols, one for incoming requests(see Fig 5.3 a) and one for self-initiated protocols(see Fig 5.3 b).

The initiated protocol starts by sending an overview message to a selected node. The listener node responds with a complement message, containing all metadata elements provided by the listener node and a list of requested metadata. If the message is not received by the initiator within a timeout period, the protocol logs the result and terminates gracefully. This happens in the case of the other node moves out of reach before a return message is sent or the overview message does not go through, hence transmission error.

Upon receiving the complement message all information is appended to the local knowledge. If the list of needed elements contains any entries, the corresponding metadata is sent to the listening node.

The listening node always keeps a ready protocol listening for overview messages. When a message is received the protocol constructs a complement message. It also adds the list of needed elements as described in Section 5.4.1.

The protocol now waits for the other node to return the needed elements. When the needed message is received or no message is received with in the timeout period, the protocol terminates. If no metadata is needed, this step is skipped and the protocol is terminated directly.

5.5.2 Broadcast propagation

In epidemic routing there is one to one communication between synchronizing parties. The one to one technique introduces many messages. To reduce this I use broadcast mechanisms to take advantage of the radio networks shared medium. Such a protocol can communicate from one to many using only one message. This is thought to introduce less overhead, especially when merging networks and when nodes are already synchronized.

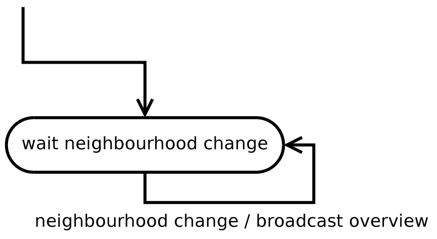
At the initiator the protocol has to be able to respond to two events. One, changes in the neighborhood or other triggers. Two, incoming complement messages. Both events have their own listener, since they need different responses, as described below.

The initiator will react to internal events such as new metadata or new neighbors by starting a new synchronization protocol as described. The broadcast protocol will then broadcast an overview message (see Fig 5.4 a). The subprotocol will then regain the waiting state. If no one responds to the message, the initiator will not follow up on this action.

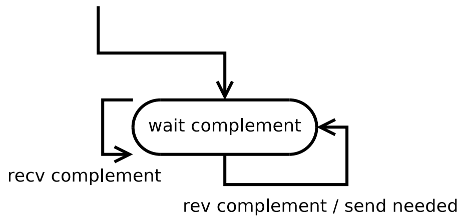
Only those listeners that receive the overview message and need some of the elements will respond, following the state machine in Figure 5.5. The response will trigger the synchronization protocol. The initiator will receive the complement message, handle the provided elements, and read the needed list to see if the initiating node is required to provide anything to the listening node. If so, the needed message is constructed and sent (see Fig 5.4 b).

The decoupling of the listening and initiation subprotocols is vital to achieve one to many communication. With this design the nodes can independently of the state on the initiator side, respond to an overview message. It also handles that many nodes respond simultaneously and that some even respond late. If there is more than one responder to a broadcast message, the reduction on the network load is equivalent to n-1 overview messages both in size and transfer time, where n is the number of response messages to the overview message.

The listening node waits for an overview message which starts the protocol. The overview is examined by the listener and a complement is constructed. The


wait neighbourhood change
neighbourhood change / broadcast overview

(a) State machine for neighborhood changes and other triggers



(b) State machine for exchanging metadata

Figure 5.4: Broadcast propagation protocol at initiator side.

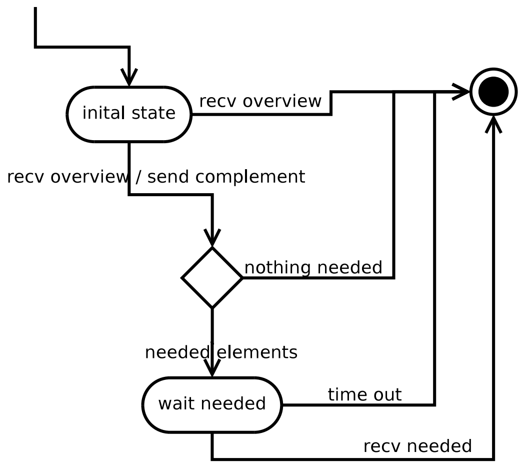


Figure 5.5: Broadcast propagation protocol machine at listening side.

complement is not sent unless either some elements are in the complement, or there exist some needed elements. Only if the complement message is sent, the protocol will continue, if not, it logs and terminates. When the complement is sent and the protocol requests some needed elements from the initiator, it waits for a needed message which transfers the requested elements. If no elements are requested, the protocol terminates. If the needed message does not return within a timeframe, a timeout occurs and the protocol terminates.

5.5.3 Semantic propagation

The broadcast propagation protocol tends to give a propagation pattern similar to that of an expanding ring search. This is expected and wanted. The effect is caused by the neighborhood scope of each node, as described in Section 3.2.3. To break the pattern and speed up the dissemination process we have developed a protocol that tries to start the propagation at multiple places at once. These points are constructed from information about groups in the network. Groups are further discussed in Section 2.4.2. Theoretically this will give greater speed of dissemination to the group members, as a side effect it gives multiple starting points for the dissemination process and results in faster dissemination to all. The downside is that this will generate more traffic by using multi-hop synchronization. This is an additional overhead, and must be handled with care and be monitored. I will try to evaluate if the benefit of multihop synchronization gives notable performance gains and what the size of the additional overhead is. This will be evaluated in later chapters.

The protocol in itself uses both the epidemic and the broadcast protocols as first and second step in the process. These are used since they already are developed and tested. They are also solving the task at hand, namely to syncronize one to one, and one to many.

The two steps of the semantic propagation are;

1. Group synchronization

2. Neighbor synchronization

The first step uses the epidemic propagation protocol (see Section 5.5.1) to perform a multihop synchronization with all the other nodes in the group, one by one. The second step uses the broadcast propagation protocol at each of the group members and continues as described for the broadcast protocol (see Section 5.5.2).

The semantic propagation introduces another level into the network, an overlay network similar to the one of a publish-subscriber network like DENS (see Section 2.8.1). The multiple start points, from each of the network members, can work in parallel by propagating new changes in the network simultaneously.

Chapter 6

Implementation

In the implementation chapter, I will describe in detail how the protocols as well as the supporting classes are implemented. This will be done by first laying out the overview, UML diagrams can be found in the appendix, later more detailed description follows together with pseudocode. The supporting classes like communication and topology information are described before the implementation of the DENS protocol is specified in detail.

6.1 Implementation of GMDM

The GMDM component handles both the dissemination process and the presentation of metadata. It synchronizes its local storage with neighboring nodes and presents the shared local storage to them using the protocols. Local storage and presentation will be handled in the metadata tracker subcomponent and the dissemination and synchronization in the SyncManagerController hierarchy which are the implementations of the synchronization protocols. The diffrent parts of GMDM will now be described.

6.1.1 Metadata tracker

The task of the metadata tracker, implemented in GmdmMetadataTrack class, is to handle tracking of metadata elements. This tracking embeds storing and retrieving metadata. This is implemented as a layer on top of the MDS local storage (MdsLs), which is a relational database adapter. The metadata tracker solves the translation of metadata elements to database tuples. By doing this, the metadata tracker gives a transparent and persistent presentation of metadata. MIDAS needs to search metadata by source node and by table name, to be able to search and retrieve data. The GmdmMetadataTrack class have functions returning results for these tasks.

6.1.2 Metadata

Metadata is represented in MIDAS as a Java class (see Figure 6.1). Each metadata element represents a table replica at a node, the metadata element is uniquely identified by a combination of table name and the node name, which is sufficient for MIDAS usage. This identifier is produced by the metadata class

```
public class GmdmMetadata implements MdsMetadataElement,
    Serializable {
  private String name;
  private String nodeId;

  /**
   * creates a new metadata element
   * @param name
   * @param nodeid
   */
  public GmdmMetadata(String name, String nodeid)
  {
    this.name = name;
    this.nodeId = nodeid;
  }

  public String getName() { ... }
  public String getNodeId() { ... }

  /**
   * returning a unique identifier of this metadataelement. is
      unique for this replica or this source of the original table
      .
   * @return
   */
  public String getIdentifier() {
    return ""+ name + "-" + nodeId;

  }

}
```

Figure 6.1: Listing of the important parts of the Metadata class

itself. The identifier is used by many functions that work with metadata, as the protocol overview message for example. Since the metadata class is the one defining the identifier, it can be changed to handle more complex features in the future. In addition to the identifier data, the metadata class can also include other data; creation date, information about whether this is a replica, where the parent table instance is located, data types and so on.

The reason for this extendable metadata class is that it represents the core of the data model. Within MIDAS there has been discussion on how to delete a table replica, and how to disseminate the unavailability of this table replica. By altering the identifier we would be able to disseminate the deletion within the same framework as we disseminate a creation. The identifier would simply change from “testtable-node1” to “testtable-node1;deleted”, which would in practice disseminate another metadata element, but with the semantics of deleting the first. Without this extendability in the metadata class such a extension would be more complex to implement.

In this thesis I have concentrated on the use of metadata in MIDAS, where the demands for metadata information is low, it is restricted to name and location. The metadata concept can, as I have discussed above, be extended to contain much more information. Use, other than the current MIDAS required metadata, is not the focus of this thesis and will therefore not be discussed or implemented further.

6.1.3 Synchronizer

The GmdmSynchronizer, also called the Synchronizer, provides an interface to the MetadataTracker specially designed for the protocol implementations. The Synchronizer handles the comparison of the two local storages under synchronization. To compare the two local storage sets set theory will be used, since the interest are of the complement of one set compared against another. These functions are hard to implement and must therefore be carefully tested. The use of the synchronizer is typically; “return the complement of this metadata list from local storage”, “given this list of metadata what can the local storage provide”, and “insert this list of possibly new metadata”

The separation of the synchronizer decouples the protocols and the MetadataTracker, additionally it separates the synchronizers functionality from the protocols, which is therefore easy to unit test. Since all protocols require the same functionality the separation combines the functionality into a separate object. The testing of the synchronizer has given a confidence to the stability and correctness of the algorithms. By confining the synchronizer logic in the synchronizer I can implement other tactics like probability based dissemination or other models of synchronization by altering the synchronizer algorithms. One can also see a possible solution to encryption by simply encrypt everything outside the synchronizer, since it forms the boundary against the network, if security is important.

In MIDAS the synchronizer its used in is basic from, doing the set theory operations on the different lists. This is also the version used during this thesis.

6.1.4 Sync manager controller

Each protocol is developed by implementing an idea often invented on a white board or such. When the implementation is due, all ideas and constraints are to be implemented into one streamlined protocol. To keep control over what is actually going on, it is important to have understandable source code. This is achieved by doing only the protocol specific actions inside the protocol implementations, all other handling, like exceptions, timeouts and anomalies, which are not part of the protocol process, has to be separated from the implementation. When this is done, the source code gives an exact representation of the protocol specification, understandable for the reader.

To build protocol implementations, I have decoupled GMDM and the protocol implementations. This is done by making a separate layer called the SyncManagerController. This layer handles the communication, sequential execution of the protocol, and hides the protocol implementation from GMDM.

Communication

To handle communication in the asynchronous fashion that is required when using MANETs, the SyncManager uses the RequestHandler, which will be described later. For now, all it does is making the asynchronous communication appear synchronous to the protocol developer. It handles timeouts and round trip delays so that the protocol only relates to sending and receiving as synchronous calls.

The protocols consist of well defined steps, which in their execution sends or receives messages. Each of these steps is required to be executed in sequence so that the communication order remains correct, since each step responds to messages from the other protocol party. The SimpleSyncProtocol (see Figure 6.2) requires for example that the SendView step finishes, which sends the overview message, and that the ReceiveCompliment step does not start execution before all subactions in the previous step are finished on the initiating node. One can say that the protocol is defined by its steps and their order. Each of the steps is implemented using the command pattern [7], which gives a common interface for how to execute the step. We can now construct each protocol by picking the steps that are required and order them in the correct way, thus forming a protocol.

Sequential execution

This decoupling of step and protocol gives the ability to test each step according to its specification, more on this in the unit test, Section 6.8.1. The ability to implement each step as an isolated action has made the complex protocol easier with respect to both debugging and development, but the sequential execution has become even more important. If steps are executed in parallel or in the wrong order, the messages will be sent in the wrong order. This might lead to deadlocks and the dissemination will grind to a halt. By letting the frame work, the SyncManagerController, ensure the sequential execution we will avoid these kinds of race conditions without any effort from the developer. The sequential execution is handled by chaining the steps by using the ChainManager.

```
   public class SimpleSyncProtocolController
 2   extends SyncManagerController
     implements MdsCrtCallBackAdapter, ChainManagerEndInterface,
         SyncManagerInterface {
 4
     public SimpleSyncProtocolController(MdsInternalInterface mds)
 6   {
       super(mds);
 8     this.mds = mds;
       super.toComponent = AbstractMdsMessage.GMDM;
10   }

12   public void respondTo(GmdmMessage msg) throws
         SyncManagerException {
       respondedTo.add(msg.getFromNode());
14
       isResponseOnly = true;
16     setRequestId((String)msg.getRequestId());

18     try {
         chain.add(new ReceiveAndHandleView());
20       chain.add(new ReceiveNeeded());
       } catch (ChainException e) {
22       MdsLogger.error(e, this);
         throw new SyncManagerException("Could not construct chain
             manager", e);
24     }

26     start();
     }
28
     public void syncWith(String toNode) throws SyncManagerException {
30
       syncedWith.add(toNode);
32     this.toComponent = AbstractMdsMessage.GMDM;
       try {
34       chain.add(new SendView());
         chain.add(new ReceiveCompliment());
36     } catch (ChainException e) {
         throw new SyncManagerException("Could not construct chain
             manager", e);
38     }

40     start();
     }
42
     public void triggerNewMetadata(String fromNodeId)
44     throws SyncManagerException { ... }
   }
```

Figure 6.2: Listing of the simple synchronization protocol, with the parts handling construction of the protocol. We can see from lines 19-20 and 34-35 how the steps are added to the execution chain.

```java
public interface SyncManagerInterface extends Runnable {

  public void respondTo(GmdmMessage message)
    throws SyncManagerException;
  public void syncWith(String nodeId)
    throws SyncManagerException;
  public void triggerNewMetadata(String fromNodeId)
    throws SyncManagerException;
  public void startSync()
    throws SyncManagerException;

}
```

Figure 6.3: Listing the interface implemented by any synchronization protocol. All protocols are implemented as SyncronizationManagers.

Interface

Each of the protocols has to respond to different situations, these situations are triggers described in Section 5.3. These include; on new metadata, both own and retrieved, and on new neighbor, represented in the protocol interface (see Figure 6.3) as functions *syncWith()* , used for new neighbors and *triggerOnNewMetadata()* , for new metadata.

In addition to the triggers we need a function for responding to other nodes' synchronization request, this function is named *respondTo()* . It handles all types of messages and is responsible for executing the correct part of the protocol, it can also throw an exception if the message is not recognized. The *syncWith()* call is used, to synchronize local storage with one specific node. *respondTo()* is used when receiving a message that is sent as an initiation of a new protocol by another node. *triggerOnNewMetadata()* will start synchronizing with all neighboring nodes, to disseminate the new information. When one of them is called, the protocol constructs the chain of steps to handle the specific request.

While constructing a protocol the ChainManager of the SyncManagerController is loaded with the steps that are to run during this protocol. This is done as soon as the protocol knows what situation to respond to. This is decided upon calling one of the SyncManagerController interface functions.

6.1.5 Chain manager

As mentioned, the protocol consists of a chain of steps. These steps are organized as an ordered set inside the ChainManager (see Figure 6.4). The ChainManager is constructed as a series of command objects, in this case protocol steps. Before the chain starts execution, the *runChain()* command is invoked. It starts the chain execution with the first step, upon completion the next step is given the go ahead. If one of the steps is aborted or throws an exception the ChainManager will stop the entire chain, log the state of the chain and run a special abort routine defined by the protocol or manager implementation. In MIDAS the most common abort routine is to log and terminate the protocol gracefully. If all steps are executed successfully, the special end routine is executed and the protocol terminates. Most successful end routines contains logging.

```
public class ChainManager {
  ChainManagerEndInterface end = null;
  private List chainElements = null;

  public ChainManager()
  {
    chainElements = new ArrayList();
  }

  public void runChain(Map configuration)
    throws ChainException
  {

      for(int i = 0; i < chainElements.size(); i++)
      {
        currentPoint = i;
        try {
          ((ChainElement)chainElements.get(i)).run(configuration);
        }catch(ChainAbortedException e)
        {
          MdsLogger.info("Chain stopped due to an abortion;" + e.
              getMessage(), this);
          return;
        }catch (ChainException e) {
          throw new ChainException("Trouble with the "+ i +" chain
              element", e);
        }

      }

      end(configuration);
  }

  public void add(ChainElement element)
    throws ChainException
  {
    chainElements.add(element);
  }

  public void abort(String message, Exception cause)
    throws ChainException
  {
    throw new ChainException(message, cause);
  }

  public void setEnd(ChainManagerEndInterface end)
  {
    this.end = end;
  }

  public void end(Map configuration)
    throws ChainException
  {
    if(end == null)
      throw new ChainException("End listener not specified", null);
    else
      end.end(configuration);

  }

}
```

Figure 6.4: Listing the ChainManager, displaying the *add()* function to add new steps called chain elements, *end()* , and *abort()* functions to use for listening for abort or successful end state. The *runChain()* starts the configured chain, running one step at the time.

Each of the steps is a separate command object. The commands often require common artefacts like session info and resources like local storage. There exists a shared pool of artefacts where interstep resources and other shared artefacts can be stored. This is provided by the command objects via the ChainManger.

6.2 Protocols

Each new protocol is constructed as a subclass of SyncManagerController. What separate the protocols are which steps are included and how each of the interface calls (*syncWith()* , *respondTo()* , *triggerOnNewMetadata()*) are handled. The SimpleSyncProtocol handles the entire protocol in one session, the BroadcastProtocol separates the protocol in two parts which gives the one-to-many feature.

As we can see both models of protocol design are supported via the SyncManagerController, while the GMDM sees no difference to which protocol is used. It merely passes the initiation message to the selected protocol factory, which instantiates a new protocol of the reconfigured type. If the node is to initiate a protocol itself, GMDM will call a factory method, which produces the correct protocol instance, where the implementation is hidden under a interface common to all protocols.

While all these supporting structures surround the protocol, the only things that are handled in the protocol implementation, are sending, handling and receiving messages. This gives a more understandable protocol source code which in turn reduces the amount of bugs introduced.

Messages

Each protocol communicates through a set of messages. All messages in MIDAS has payload either in the form of a text message or metadata. All messages are constructed over a super class GmdmMessage which uses the implementation of an MdsMessage. By using this hierarchy of messages each new layer handles only its additional function, and uses the support provided by other components and layers. The protocol messages are required to have a type and to contain payload. The type is used to identify the state of the protocol and semantics of the message, the payload is used to exchange metadata.

Overview message Messages in GMDM protocols are separated into three groups: overview, complement and exchange messages. The overview message delivers a small representation of the local storage at the sender. This representation is as small as possible, but not smaller than we can identify each element uniquely. Currently this is the identifier function from the metadata class. The metadata identifier strings are put into a Vector list inside the message. A Vector is a Java implementation of a dynamic array.

The overview message has the semantic meaning of starting a protocol.

Complement message This message is sent as a response to an overview message, the metadata is the complement of what was presented in the overview versus the local storage, in other words: the elements stored in the local storage but were not present in the overview. Sometimes this is referred to as the provideable metadata. The entire metadata object is sent, to move additional metainformation.

The metadata objects are contained in a Vector, and the requested metadata identifier strings are kept in another Vector.

Exchange message Exchange message is as simple as the overview message. It carries only a bag of metadata objects. Each object is the full version metadata, thus this message can be large in size. The message is made to transport metadata from one node to another.

6.2.1 Simple synchronization protocol

As the names say, this is the simple implementation of dissemination. It bases its approach on epidemic routing, taken from routing principles. The mechanisms of epidemic routing are described in Section 2.4.2. The implementation will serve as a reference for optimalization regarding the other two protocols and as the simple, strait forward implementation.

It is implemented as a one to one protocol, meaning that it handles only one sender and receiver. The protocol exchanges overview to establish difference and then the complement message transfers the actual elements back to the initiator together with the list of needed elements. The exchange message is used to exchange the needed elements.

The messages are controlled by the steps that the simple synchronization protocol is consistent of;

- SendView (see Figure 6.5), sends the overview message. Located on the initiator side.
- ReceiveAndHandleView (see Figure 6.6), receives the overview and handles it, by sending a compliment message. Located on the listening side.
- ReceiveCompliment (see Figure 6.7), receives the compliment and sends the optional exchange message. Located on the initiator side.
- ReceiveNeeded (see Figure 6.8), receives and inserts any elements from the exchange message. Located on the listening side.

Each step is describe using pseudocode. The SendView step is described by listing the actual java implementation, to give a view of the actual work needed to implement a step. The implementation of the SimpleSyncmanager implementing the protocol interface is described in Figure 6.9

No interception is done to prevent redundant protocols, which means that when the protocol is set into action it finishes even though there are no differences between the two nodes. However, no exchange message will be sent unless there is content inside. This makes the last message optional.

Since this protocol does not use smart tricks to speed up the protocol it will probably generate a lot of overhead traffic and processing on all nodes involved.

6.2.2 Broadcast synchronization protocol

To take advantage of the knowledge learned from developing the simple synchronization protocol together with studies of routing protocols in MANETs

```
public class SendView extends ChainElement {
  GmdmMetadataExchangeMessage exchange = null;
  public void run(Map configuration) throws ChainException {
    RequestHandler handler = (RequestHandler)configuration.get("
        node");
    MdsInternalInterface mds = (MdsInternalInterface)configuration.
        get("mds");
    GmdmSyncOverviewMesasge message  = new GmdmSyncOverviewMesasge
        ((String)configuration.get("toNode"), mds);
    Vector knownElements = new Vector();
    Iterator it;
    try {
      it = mds.getGmdm().getSyncronizer().getOverview();
    } catch (GmdmException e) {
      throw new ChainException("could not get overview" , e);
    }
    while(it.hasNext())
    {
      String element = (String)it.next();
      MdsLogger.debug("element in overview: " + element, this);

      knownElements.add(element);
    }
    try {
      message.addAllKnownElement(knownElements);
    } catch (GmdmException e) {
      throw new ChainException("Could not add all known elements" ,
          e);
    }
    configuration.put("overview", message);
    MdsLogger.info("sending SyncOverview message to:" + message.
        getToNode(), this);
    handler.send(message);
  }
}
```

Figure 6.5: Listing of the class SendView which implements a step.

```
def run:
  overview = requestHandler.receive();
  exchangeMessage  = new ExchangeMessage(overview.getFromNode(),
      mds);
  exchangeMessage.addNeededElements(synchronizer.getNeededElements(
      overview.getAllKnownElements()));
  exchangeMessage.addMetadataElements(synchronizer.getCompliment(
      overview.getAllKnownElements()));
  requestHandler.send(exchangeMessage);

  if(exchangeMessage.getNeededElements().isEmpty())
  {
    throw new ChainAbortedException("No needed elements",null);
  }
```

Figure 6.6: Pseudocode of the ReceiveAndHandleView step

```
def run:
  complement = requestHandler.receive();
  if not (complement.getAllElements().isEmpty())
  {
    synchronizer.instertSyncronation(metaData.getAllElements();
   Gmdm.newMetadataReceived(metaData.getFromNode());
  }

  if not (complement.getNeeded().isEmpty())
  {
    metadataExchange = new GmdmMetadataExchangeMessage(metaData.
        getFromNode(), mds);
    neededMetadata = synchronizer.getElementsByIdentifyer(
        complement.getNeeded());
    metadataExchange.addMetadataElements(neededMetadata);
    requestHandler.send(metadataExchange);
  }
```

Figure 6.7: Pseudocode describing the step ReciveComplement

```
def run:
  neededMetadata = requestHandler.receive();
  if not (neededMetadata.getAllElements().isEmpty())
  {
    mds.getGmdm().newMetadataReceived(metaData.getFromNode());
  }else
  {
    MdsLogger.debug("received empty needed message from " +
        metaData.getFromNode(), this);
  }
```

Figure 6.8: Pseudocode describing the ReciveNeeded step

```
public class SimpleSyncProtocolController
  extends SyncManagerController
  implements MdsCrtCallBackAdapter, ChainManagerEndInterface,
      SyncManagerInterface {

  public SimpleSyncProtocolController(MdsInternalInterface mds)
  { .. }


  public void respondTo(GmdmMessage msg) throws
      SyncManagerException {
    ...
    chain.add(new ReceiveAndHandleView());
    chain.add(new ReceiveNeeded());
    ...
    start();
  }

  public void syncWith(String toNode) throws SyncManagerException {
    ...
    chain.add(new SendView());
    chain.add(new ReceiveCompliment());
    ...
    start();
  }

  public void triggerNewMetadata(String fromNodeId)
    throws SyncManagerException {
    ...
    for(int i = 0; i< nabours.size(); i++)
    {
      ...
      mds.getGmdm().getSyncManager().syncWith(nodeId); //
          constructing a new manager
      ...
    }

  }
}
```

Figure 6.9: Pseudocode describing the vital parts of the implementation behind the protocol interface when implementing SimpleSyncManager. “...” indicates code removed to enhance readability.

and the radio transmission medium such as wifi, we developed an optimization. This optimization tries to handle some of the major drawbacks from the simple protocol by also using some other techniques.

When developing the epidemic synchronization protocol, I detected some optimization gains that would improve the performance of the protocol. These are the work pattern of the epidemic routing, when going in a one-to-one synchronization with each of the nodes in its neighborhood. This is both costly for the network and time consuming. What I observed was that the overview message sent each time, was the same. By reusing the bandwidth used by the first overview message would we save bandwidth.

I had to make the protocol one to many to make this possible. I have handled this by sending one broadcast overview message, this is not sent via unicast but broadcast. By sending a message in a radio medium, all in reach will hear and receive the message. Only by rule and conduct will it not be read by others than the recipient. When using the broadcast address, everyone is the recipient, so while all hear the message in the first place, I take advantage by letting all use the message. With 10 neighbors this is a saving of 9 messages in the first step.

I also observed that there were many nodes that did not request any elements from the overview, since they already where synchronized. These nodes were, by protocol, forced to send an empty message. To handle this, I made the complement message optional. We now serve all the complement messages that are received. For each message that comes in, we store the metadata and transfer the needed metadata via an exchange message. This means that we in general can stop a protocol at the receiver of an overview message. We still piggyback any request for needed elements, but only if the complement message is to be sent.

Figure 6.10 describes the implementation of the SynchronizationManager. The steps BroadCastOverview described in Figure 6.11, initiates the entire protocol. On the receiver side is the steps ReceiveBroadcastAndHandleView (see Figure 6.12) and ReceiveDeliveryMesasge (see Figure 6.13) located. The initiator still need to offer its metadata elements to any listener this is done in the step HandleExchangeMessage described in Figure 6.14.

6.2.3 Semantic synchronization protocol

As we shall see in Chapter 8 the broadcast protocol show good performance when dealing with dense clusters and dissemination to neighbors. We want to see if there is techniques that give higher dissemination speed in larger topologies, with more neighbors per node, like the grid scenario.

We want to implement a type of highway for metadata that bring new metadata to multiple places fast, and then fall back to the normal dissemination. This might stop the delay we have seen for dissemination through multiple hops, where nodes far away are naturally getting the metadata information late. To cope with this, we implement groups of nodes, which first synchronized with all other in its own group, then start dissemination in the normal one hop way. These groups can in real life be formed from roles or reporting chains dynamically, as described in Section 5.5.3. The groups in our implementation are predefined in a XML file.

The protocol has two stages. The first is to synchronize with its own group, regardless of communication hop count. All nodes in reach are initiated with

```
public class BroadCastSyncController extends SyncManagerController{

  public BroadCastSyncController(MdsInternalInterface mds)
  { ... }


  public void respondTo(GmdmMessage message) throws
      SyncManagerException {
    if (message instanceof BroadCastOverviewMessage) {
      ...
      chain.add(new ReceiveBroadcastAndHandleView());
      chain.add(new ReceiveDeliveryMesasge());
      ...
    }else if (message instanceof ExchageRequestMessage) {
      chain.add(new HandelExchangeMessage());
      ...
    }
    start();
  }

  public void syncWith(String nodeId) throws SyncManagerException
  { syncWithEveryOne(); }

  public void triggerNewMetadata(String fromNodeId)
      throws SyncManagerException
  { syncWithEveryOne(); }

  public void startSync()
    throws SyncManagerException
  { syncWithEveryOne(); }
  private void syncWithEveryOne()
    throws SyncManagerException
  {
    ...
    chain.add(new BroadCastOverview());
    ...
    start();
  }


}
```

Figure 6.10: Listing of the BroadCastOverview Syncmanager implementing the broadcast protocol. The “...” indicates code removed for readability.

```
def run:
  //BroadCastOverviewMessage sets its to address to the broadcast
      address of the network interface
  overviewMessage = new BroadCastOverviewMessage(mds);
  overviewMessage.addKnownelements(synchronizer.getOverview());
  requestHandler.send(overviewMessage); //broadcasting the message
```

Figure 6.11: Pseudocode describing the BroadcastOverview step.

```
def run:
  overview = requestHandler.receive();
  complementToOverview = synchronizer.getCompliment(overview.
      getAllKnownElements());

  exchangeRequestMessage = new ExchageRequestMessage(overview.
      getFromNode(), mds);
  exchangeRequestmessage.addRequest(synchronizer.getNeededElements(
      overview.getAllKnownElements()));

  exchangeRequestmessage.addProvided(complementToOverview);

  if(message.getRequest().isEmpty())
  {
    //save the wifi net, and abort the protocol here
    MdsLogger.info("No info needed from " + message.getFromNode(),
        this);
    throw new ChainException("Aborted due to non needed", new
        NotMoreElementsNeededException());
  }

  requestHandler.send(exchangeRequestmessage);
```

Figure 6.12: Pseudocode describing the ReceiveBroadcastAndHandleView step.

```
def run:
  neededMetadata = requesthandler.receive();
  synchronizer.instertSyncronation(neededMetadata.getData())
```

Figure 6.13: Pseudocode describing the ReceiveDeliveryMessage step, which receives any requested elements.

```
def run:
  exchangeMessage = requestHandler.receive();
  synchronizer.instertSyncronation(exchangeMessage.getProvided());

  if(exchangeMessage.getRequest().size() > 0)
    {
      deliveryMessage = new DeliveryMessage(exchangeMessage.
          getFromNode(), mds);
      deliveryMessage.addData(synchronizer.getElementsByIdentifyer(
          exchangeMessage.getRequest()));

      requestHandler.send(deliveryMessage);
    }
  }
```

Figure 6.14: Pseudocode describing the ReceiveExchangeMessage step.

```
public class SemanticSyncController extends SyncManagerController{

  public SemanticSyncController(MdsInternalInterface mds)
  { ... }

  public void respondTo(GmdmMessage message) throws
      SyncManagerException {
    if(message instanceof BroadCastOverviewMessage)
    {
      controller = new BroadCastSyncController(mds);
    }else if(message instanceof GmdmSyncOverviewMesasge)
    {
      controller = new SimpleSyncProtocolController(mds);
    }else if( message instanceof ExchageRequestMessage)
    {
      controller = new BroadCastSyncController(mds);
    }
    ...
    controller.respondTo(message);
  }

  public void syncWith(String nodeId) throws SyncManagerException {
    ...
    simpleSyncProtocol.syncWith(nodeId);
  }

  public void triggerNewMetadata(String fromNodeId)
      throws SyncManagerException
  { startSync(); }

  public void syncWithGroup()
  {
    ...
    for(int i = 0 ; i < groupMembers.size(); i++)
    {
      ...
      syncWith((String) groupMembers.get(i));
      ...
    }
  }

  public void syncWithNeigbours()
  {
    ...
    BroadCastSyncController bcast = new BroadCastSyncController(mds
        );
    bcast.startSync();
    ...
  }

  public void initializeGroup()
  { ... }

  public void startSync()
  {
    ...
    syncWithGroup();
    syncWithNeigbours();
  }

}
```

Figure 6.15: Listing the SemanticSynchronization protocol. The "..." indicates code removed for readability.

the simple protocol one by one. The implementation is described in detail in Figure 6.15. To route the packets to its right recipient, we need the total topology information. We have used OLSRD for both topology and routing in the semantic protocol.

The second stage is for each of the group members to start dissemination, by starting a broadcast synchronization with its neighbors. This gives the dissemination multiple starting points. The idea is that this gives us a higher efficiency, especially inside the group.

All these dissemination processes, starting at the group members, goes in true parallel, the technique will hopefully have higher speed with less overhead. This is yet to be seen, and will be discussed in Chapter 8.

One future optimalization is to let the intermediate nodes listen in and sniff up the information that is relayed through them. By doing this, we can make even more start points for the metadata dissemination. This sniffing technique is, however, not implemented while this would require implementing a cross layer mechanism to fetch or sniff the packets that is part of the semantic synchronization protocol from the transport layer. Since the packets are routed by the transport layer, helped by the linux kernel and the OLSRD routing daemon, are they out of reach for Java on the intermediate nodes. The focus is directed towards the multiple starting points, and the vision of this improving the dissemination speed.

6.3 CRT implementation

CRT handles communication and routing in the MIDAS middleware, it is developed by another partner of MIDAS, Telefonica in Spain. While this partner did not focus on MANETs as their primary development goal, there was no working CRT implementation which fulfilled our needs, at the time of the experiments. This caused many problems for the implementation of GMDM and MDS. We then discussed how to work around this problem; we needed communication, routing and topology information services. There were no suitable open source projects, as we have high demands on what CRT is going to deliver. Another aspect is that MIDAS is implemented in Java, most routing protocols are not. We also decided that controlling the inner workings of CRT would benefit the development of our own components.

We went forward and started our own implementation of the CRT component. The MIDAS partners had already agreed upon the interfaces for all components, including CRT. This meant that our job was to implement behind this interface, in a way that handled MANETs, and met our needs.

I used quite some time working out the requirements and how to accomplish them. The requirements are three-fold; communication (point to point), routing (end to end), and topology information (neighborhood info).

6.3.1 Communication

The communication component has to solve the same task as we expect from MIDAS CRT, which is to send and receive delay tolerantly in a MANET, and give routing information. The architecture defined that the sending of information should be best effort and delay tolerant.

I implemented an asynchronous communication model, where one can send a message to a recipient, but there will be no response to whether or not the message is received. Any mechanisms for checking that messages come through, must be implemented in the above layers, and is naturally a hard task in the delay tolerant networks. This decision was implemented as UDP traffic over the wifi network using Java sockets. However, delaytolerant message passing is not implemented since none of the scenarios did require it.

The CrtCommunication, which is the lowest level, handles the UDP sockets and actual datagram network packages. CrtOverUdp handles the MIDAS to CrtCommunication translation, which embeds translation to and from the IpMessage format, which MIDAS can use. The goal for this two-layered approach is the ability to change socket and transport protocol behind the interface by changing the implementation of CrtCommunication, without changing the CrtOverUdp layer. The names should have been switched to indicate that the communication is using UDP and the above layer is only relating to IP packets.

The CrtOverUdp object is always running and listening for packets through the interface from CrtCommunication. When a packet arrives to the node, CrtOverUdp will handle the translation from Datagram (network packet) to IpMessage (Java object) and send the packet up the layers to the object that is registered for listening. This is in the case of MIDAS the CrtAdapter.

CrtOverUdp naming scheme

The CrtOverUdp implementation of the CRT interface is exactly what the name implies. It is the CRT implementation using UDP the transport protocol. This name is selected to show the nature of the implementation. The CRT interface contains both transport layer functions and topology functions. This implies that the CrtAdapter implementing the entire interface needs to implement both transport layer functions, send and receive, and the topology manager functions, *getNeighborHood()* . The CrtOverUdp handles the transport layer functions, by using CrtCommunications to actually do the UDP sockets. However could the CrtOverUdp be called CrtTransport and CrtCommunication be called CrtUdpMessageing. In this way would the names reflect the implementations underneath. This would also enable changing the transport layer implementations to other than UDP with limited consequences. As of today, the CrtAdapter can not do so with out changing names of the transport layer component. These are minor changes but will make the maintenance and further development easier.

Topology functions is implemented by the OlsrInt, which is the OLSRD integration component.

Abstraction

To be able to later use the official CRT developed by MIDAS, I abstracted the use of the CRT component. This is done by adding an additional layer in-between the MDS and the CRT component, the layer is called CrtAdapter. In this way we could use different implementations of CRT even if there were interface and usage changes in the different CRT implementations.

The CrtAdapter has the important task of routing messages and give topology information. The reason for having this service so high in the layers is that it has been changed a lot during this project. It started out as a simple implementation with no routing and our own broadcast probing service, to establish topology information. It is later changed to use the OLSRD routing daemon for both routing and topology. This layer was the lowest common ground for these changes.

6.3.2 Topology manager

To give our CRT implementation a topology service, I implemented a BroadcastResponder. This is a probing service, which uses broadcast packets. From the responses, we are able to construct the one-hop topology.

Protocol On a regular basis, broadcast HELLO packets are sent. Everyone that receives a HELLO packet is required by the protocol to send the packet back as a response. To do this we turn the direction of the packet around, setting the sender as receiver and placing ourselves as sender. This marks the packet a HELLO response, which signals, upon receiving, a topology connection. The BroadcastResponder does not handle multihop routes or topology, it merely checks which neighbors are in one hop contact with the current. While developing and testing the BroadcastResponder, we found it to be very intrusive by generating a lot of traffic. To handle this, I made some optimizations like delaying the sending of HELLO packets until a time period has expired without any hearing other responses.

For BroadcastResponder to notify its listeners of a new neighbor, it should receive two consecutive HELLO responses. This is also required for a node to disappear, it has to not respond to two consecutive HELLO packets. This strategy is inspired by the OLSR standard where this strategy is used. This resulted in a more stable topology, and less network overhead. After using our own BroadcastResponder for a while, we found it to be not good enough, and started to look into using the OLSRD [17] implementation.

Olsr integration The integration of the OlsrInt was done by Matija Puzar, a PhD student at IFI also working in the MIDAS project. What the OlsrInt does, is to listen for messages sent from the routing daemon. The routing daemon is patched so that it sends messages in a known format on a socket. These messages describe changes in the topology of types “hop changes”, “node disappearing”, “node reappearing” and “new nodes”. From this information the OlsrInt constructs and maintains the topology in a table at the local storage.

We needed in addition a listening service from the routing daemon, like the one we had in the BroadcastResponder. I therefore implemented a smart notification service that notifies all the listeners upon a truly new neighbor. What defines a truly new neighbor is that there is a new route reported from OLSR that is one hop in length, and that is not already known. The list of known one hop routes is maintained so that nodes going away, get reported upon reappearing. We also had to implement a query into the routing information in local storage to deliver the *getneighborHoodInfo()* functionality in the CrtAdapter interface.

As one can see in the Figure 6.16, the CrtAdapter is connected to both the CrtOverUdp and the two different topology implementations, OLSR and

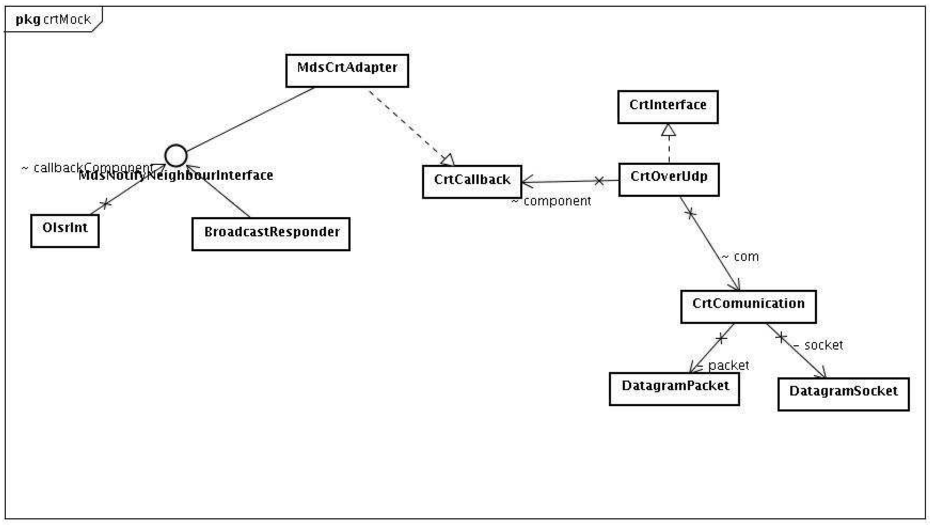


Figure 6.16: Uml diagram of the CRT implementation. Implemented to deliver Crt services communication routing and topology information

BroadcastResponder, via interfaces. The implementation is selected by the CrtAdapter, at startup it registers with the currently selected implementation, and via the interface gets called every time a new neighbor appears. This decouples the implementation from the CrtAdapter and improves cohesion, which gives us the ability to change the implementation of the topology managers without changing the CrtAdapter. Cohesion reflects the amount of focus an object has. By implementing the listener pattern in CrtAdapter, we will be able to increase the focus on the adapter job, namely to make the underlying implementation transparent.

6.4 The MDS facade component

The GMDM component is presenting a unified interface to all the GMDM features and functions. It provides, for example, functions that the QA component can use to search the metadata storage, and register new tables. These functions are implemented in the MetadataTracker, but this is made transparent by GMDM. This makes a combined facade to the GMDM sub components, hence following the facade pattern [7].

GMDM also listens for topology changes, and handles new neighbors by starting the correct protocol. The listening is essential to make the GMDM responsive to topology and local storage changes. GMDM also organizes all available protocols by using the factory pattern, which produces a new instance of the protocol upon invocation. This pattern makes it easier to change the type of protocol used, and provides control of how to construct the protocols. By hiding the creation of the protocol, we can implement logic on how to create protocols and which implementation to use. The hiding makes this logic transparent to the GMDM. The hiding is done by use of the factory pattern, by making a function designated to return new instances of a specific interface.

The MDS facade is one of the MIDAS components, and is therefore the facade that MIDAS sees. This is done by hiding all the MDS subcomponents under the interface defined for MDS. No one outside MDS can access the subcomponents. Part of this facade takes care of internal message routing, this is the messages coming from CRT addressed for MDS component. This is internal MIDAS messages, which the MDS facade gets notified about from CRT through the callback technique. The MDS facade is handling each message internally and routing each message to the correct sub component.

The MDS facade exposes an internal interface to all subcomponents, this interface serve as an internal service exchange that subcomponents can use to exchange services. The exchange handles references to the subcomponents, which is used, for example, by QA when requesting table name to node name resolution from GMDM.

MDS is the term used for the combination of all sub-components. However, we need a facade to unify the subcomponent functionality, like the ones described above, to one single component the MDS facade, which is a implementation of this facade.

6.4.1 Message passing

All network packets are received by the CRT component, it has to route the messages to the correct middleware component. This is done with the help of a field in the MIDAS message. Each of the components and subcomponents are responsible for handling messages addressed to that component.

To handle multiple subcomponents under MDS, that are going to receive messages from the outside, we needed some mechanism to route the messages to the correct subcomponent. The MDS facade gets messages addressed to it from CRT. This is done via the callback or listener system. For MDS to route the messages correctly, it needs to know - based on the semantics of the message, or on some field inside the message which subcomponent is the recipient.

I have implemented an extension of the MidasMessage to handle the MDS subcomponent addressing. This has an abstract Java class which must be further extended by any of the subcomponents in MDS that need inter-node communication. The MDS facade can upon receiving a message, check for the message type and route to the correct subcomponent via the subcomponent field. Since both GMDM and QA has session-like behaviour in their protocols, they need to address the correct instance of their protocols. This inter-protocol communication requires that the instance of the protocol can be addressed directly. This is essential with regards to protocol state and type, which is only known to the instance of the protocol and not the parent component. This is handled by the RequestHandler system.

6.4.2 Request handler

The request handler solves the problem of protocol to protocol communication. It offers a session concept which can be used in protocol implementations. It also handles many other tasks that are inherently hard within MANET communication. To solve the protocol instance to protocol instance, the protocol implementation uses a RequestHandler to send and receive messages. The RequestHandler registers itself with the MDS, getting all messages with the session id directly to itself. This ensures that all messages addressed to the protocol get routed correctly. The addressing is made up of the MDS subcomponent name eg. GMDM and a session key, which is defined by the RequestHandler upon creation. The session key must be unique within the subcomponent, to avoid session key collisions.

The RequestHandler is made to make life easier for the developer, this is done by giving the developer of protocols a suite or framework to work with. The suite consists of the addressing scheme, along with sending and receiving of message using a synchronous interface. The latter point is harder than it sounds. It embodies using the asynchronous network and making it transparent. The RequestHandler configures messages passed through its interface, so that each protocol or session is isolated from each other, it is done so transparently to the developer.

To handle the transparency of synchronous networking, I have implemented the RequestHandler's send and receive functions as blocking calls. This means that if we from within a protocol invokes the RequestHandler's receive function, the function call will not return until there is a message for this RequestHandler. The call will throw a TimeOutException if no message is received within a set timeout period. Sending and receiving of messages are implemented with a concurrent message queue. This means that there can arrive multiple messages, and the protocol implementation can handle them when ready. Unhandled messages are queued up. This feature will once again improve the development process because no messages disappear after being received at the recipient, and all messages are received in order. All this without work from the protocol developer.

To configure the sending of messages, the protocol uses the request handler's receive and send methods. These alter upon invocation the messages with the correct session id. This means that the request id is totally transparent to the protocol if the RequestHandler is used properly.

The RequestHandler was hard to develop, much effort was used, but we have had little or no problems with communication after it was finished. This is partly due to extensive testing and the following ability to verify our implementation. A mock implementation was made together with the RequestHandler that in unit test situations can be replaced with the actual RequestHandler. The mock request handler differs since it never sends or receives any message. It appends all messages that comes through the send function to the send list, and upon every receive call it takes one message from the receive queue. This enables use this mock implementation to test one protocol step, and after it finishes we can assert the messages in the queues. The send queue will reflect the messages send by the protocol, and the receive queue is the input to the protocol. We can from the input, the receive queue, and the output, the send queue and internal protocol state, evaluate if the protocol behaving correctly.

6.5 Rolle Player

The MIDAS application is only a middleware. It can run and maintain its inner state without any applications on top. To test how the middleware is working, we need to use it from a user application's point of view. While running the middleware in a test environment, we want the actions performed on the

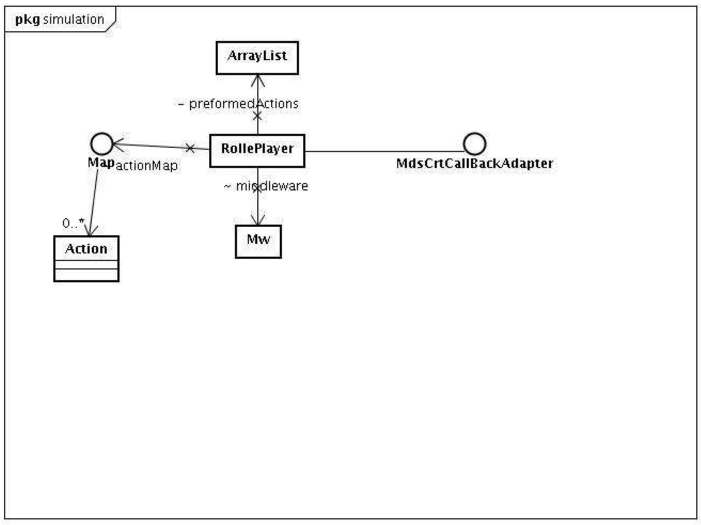


Figure 6.17: Uml diagram of the simulation application. Showing the MIDAS middleware, the separate communication and the actions.

middleware to be repeatable and manageable. This is why the RollePlayer is developed. What it does is to start up itself, read a script, initialize the actions mentioned in the script for this node, and connect to the middleware. It differs from a normal application by acting upon UDP messages sent via a separate communication channel from a control unit. These messages instruct the RollePlayer on what to do. When the RollePlayer gets a cue the action for this cue is executed according to the script. The actions are subclasses of a special command pattern class which is specified in the Action interface. The Actions contain an id, which identifies a cue, and an execution method to run upon the cue. The script is an XML file mapping the action ids to the nodes that is supposed to run them. If we want to only have a subset of all the nodes executing an action, or all the nodes, this can be specified in the XML file. The cue messages are sent from the emulator GUI to the emulator server which tunnels them into the emulator. This tunnelling is also especially developed and is a standard UDP tunnel implemented in Perl. It is implemented as is a small program listening on the defined port, any message which gets to this port is read and the internal emulator IP address is fetched. Based on the IP address a new packet is created and sent through the emulator to the correct node.

As we can see from the diagram in Figure 6.17 there is an ArrayList of performed actions, there are the actions performed by the node to keep track of what is done for post analysis. We also see the external communication channel, which is another instance of our CRT implementation, but using another port than the one the middleware is using, this is the separate communication channel. The Map is used for easy access to the Action pool.

My experience with running experiments on the MIDAS platform using the RollePlayer is very good. We can achieve exactly the same behaviour by the application for every test run, which is essential when comparing different implementations of protocols or other middleware parts.

6.6 DENS implementation

During the summer of 07 I collaborated with the DMMS research group (this is the same group as the one that runs the MIDAS project) on implementing the DENS publish subscriber protocol. The implementation is done to get preliminary test results into the functionality of the protocol. DENS is not related to MIDAS, but the protocol uses hirarchies similar to the ones found in the semantic synchronization protocol, and is serving as a study of node hirarchies and overlay networks. DENS therefore is a candidate for implementing the SubscriptionManager(SM) subcomponent of MDS. However, the interesting subject for me implementing this protocol is the distributed negotiation done to establish such a hirarchy. This work and its results is part of the paper [19].

The implementation is done following the DENS protocol described in the article [19]. It describes the different node types, and how they will react upon different events. The vision of the protocol is efficient dissemination of subscriptions in a publish subscriber environment.

DENS is an overlay network where there are two types of nodes, mediators and members. The mediators form the overlay network, which amongst themselves exchange subscriptions and messages. For a member to subscribe, it contacts any mediator, which disseminates the subscriptions to the other mediators. When a member publishes a message, this gets sent to all mediators that have one or more members that are subscribers to this kind of message. Each mediator will relay the message to the members that are subscribing.

As partitions can merge and separate, the set of connected mediators can change over time. The mediators will therefore synchronize their subscriptions to always keep them up to date in the current partition. This is done according to the protocol described in [19].

There are three protocols in DENS, I will give a short presentation of how they work;

- Mediator discovery. Every node which is a mediator, sends periodically mediator announcements. These are heard by nearby nodes and mediators. All nodes receiving announcements from this mediator, becomes its members.

  When partitions merge each mediator will check if it is the largest mediator (done by selecting the largest node ID) in its own previous partition, if it is, it assembles a REP_BROADCAST message with the mediators it represents. The representative has the task of cooperating with the encountered partition, by listening for other mediators' REP_BROADCAST messages, these newly encountered representatives start the global synchronization protocol.

- Global Synchronization. Each of the mediators that are representatives of their old partition enters the global synchronization protocol, where the lexicographically largest representative among the representatives takes the role of coordinator.

  The coordinator starts the protocol by sending a SYNC_C message to all other representatives. It contains the subscriptions known to the coordinator, hence in its own partition. Each of the mediators that receive the message sends a SYNC_REP with its contributions of further subscriptions to the coordinator, and saves the additional information from the coordinator. When the coordinator receives all SYNC_REP messages coming from all the representatives it will construct a SYNC_TOTAL message that is sent to all representatives with all elements that are discovered during this synchronization. The representatives store the information from the SYNC_TOTAL and go on to the local sync phase.

- Local Synchronization. Each of the mediators that were not part of the Global synchronization protocol will also need the information which their representative has gained through the global synchronization protocol. This is done through the local synchronization protocol. One message is sent from the representative to all mediators in its orginal partition, it contains all new subscriptions. Which are now stored on every mediator node.

After the local synchronization protocol is finished the new mediators will be merged into the old partition. This is done by the local synchronization protocol semantically signalling a merge event. Now the partition contains all mediators in range. The new representative in the next merge will be the same as the coordinator of the global sync, the largest mediator node ID.

6.6.1 Architecture

The architecture used in DENS (see UML diagram in Figure D.1) resembles that of the MIDAS system. It is component based, where communication and topology information is split into two subcomponents. What differs most is that DENS has much less components than there are in MIDAS. The DENS application is more focused on only information sharing. DENS is also reliant on OLSRD for its topology manager. However, the logic coordinating the entire application is an event model. All controllers are responding to certain events that are sent from other parts of the application. By studying the UML diagram in Figure D.3, one can see the events used in the application. These model the same events that are described in the protocol.

The different protocols are started upon certain events, and produces events upon successful completion. Produced events are fed back to the NodeController which can handle them as required. In this way the responsibility of starting the protocols in correct order, is moved from the protocol implementation to the controller, in this case the NodeController.

One of the major challenges with implementing the DENS protocol is the underlying partition manager, called the ClusterController. The DENS system requires to identify situations in the partition, these situations will be modeled as events. The ClusterController can issue events like;

- the stable event, which triggers upon a merge of two partitions, were the new partition has stabilized. The stable event triggers all needed protocols.

- the merge event, is triggered upon finishing all needed protocols issued by the stable event. When the event is triggered the two partitions will be ready for merging.

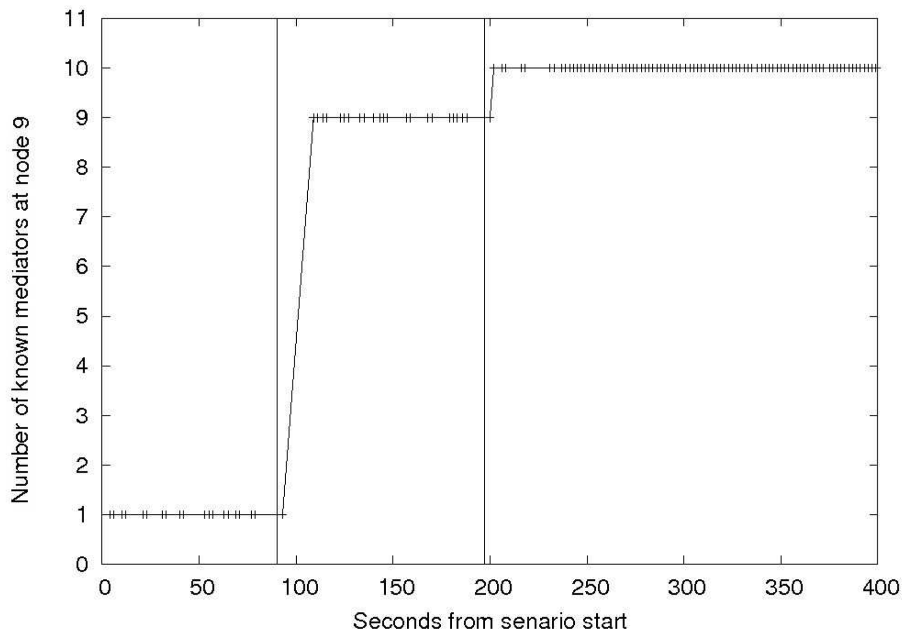


Figure 6.18: Number of mediators viewed by node number 9 in our scenario

- finishing of the global synchronization process, which should start the local synchronization, according to the protocol.

To generate the events based on topology changes, the ClusterController interface is implemented in the Olsr component. This component has responsibility of getting every changes from the partition topology. By using these events can the system calculate if an stabile event has occured. The calculation is in this implementation very simple, and uses only a timer if no changes occurs before it runs out will the stabile event occure, upon any mergeing or parting of nodes the timer will be reset. A more advanced solution is required to see the full potential of the resource managemen as dissucsed in [4].

6.6.2 Test case

The test scenario used to test the implementation resoruce-management consists of 10 nodes where all have the role of mediator. As an initial stage, eight nodes are in one partition, after 90 seconds node number 9, enters the partition. The initial positions are shown in Figure 6.19 The stable event occurs, followed by a global synchronization and local synchronization. Node 9 is now merged into the partition, and now knows nine other mediators. This will be verified by the results and shows the success of the proof of concept application. After 190 seconds node 10 enters the partition, it starts the synchronization protocols and is merged into the partition, the node we are following on the graph in Figure 6.18, is now aware of the new mediator since it knows of ten mediators. This is also the total number of mediators in the scenario which shows that the event model and synchronization protocols are working as described.

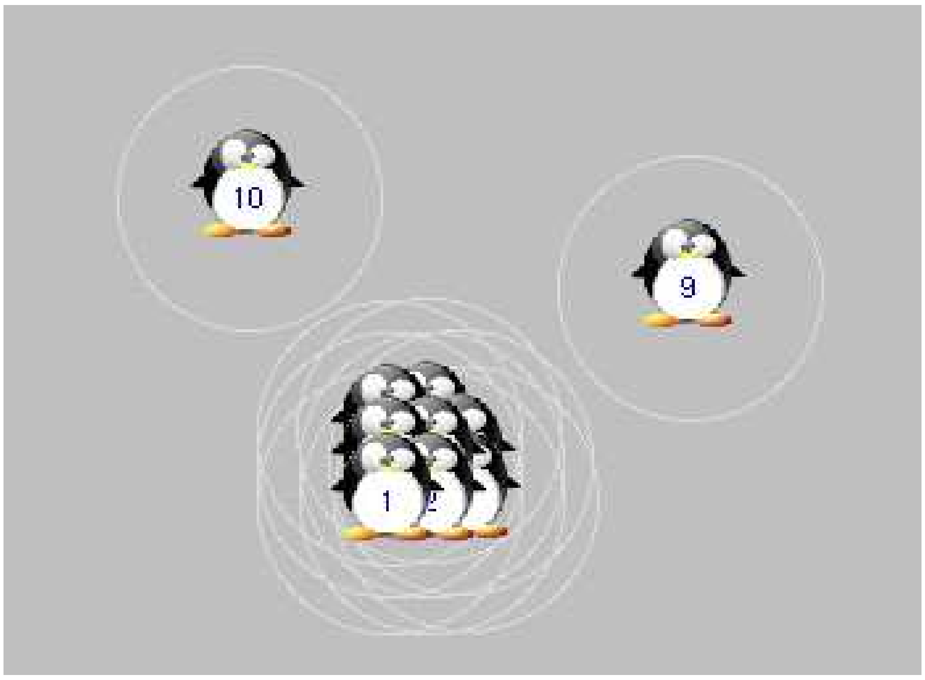


Figure 6.19: The initial topology in our scenario, showing nodes number 9 and 10 located outside the partition

6.6.3 Result

We have shown through a proof of concept implementation that the DENS protocol works, and that stable events lead to exchange of subscription data. This is shown in the graph in Figure 6.18, by looking at the increase in known mediators by node 9, and assessing that the shown graph is correct according to the protocol behaviour.

6.7 Logging

Logging is used for two purposes in MIDAS; one, to be able to monitor what is actually going on inside the middleware, and two, for post analysis of performance and state at each node. The first feature is used even when the middleware is in normal use, e.g. in production, it is the basis for debugging and error detection. The second feature is used by external tools to check on the middleware and view the state of each node with the intent of evaluating overall performance or display status.

6.7.1 log4j

log4j is an open source framework for logging, we use this framework for our debugging logs. log4j is an Apache Software Foundation [1] project. It is one of the many log4* style frameworks, and is therefore familiar from other languages and thus is easy to use. It also supports six different levels of logging, which gives us the ability to produce the correct granularity of logs both when running test scenario and in production. One can also turn off the logging from a configuration file, thus not requiring recompilation. It is reported that a disabled logging statement takes about 5 nanoseconds on an 800MHz machine [1]. This feature is essential for selecting logging frameworks, as we in production want fast execution without rewriting the source.

With the high level of configuration and the fast execution of disabled logging statements, log4j has given us what we required. It can give detailed information into protocol and routine behaviour, but only when needed. In some situations we only required to get information, warnings and errors, which are the three highest levels of logging. This reduces the amount of logs produced, but still gives us the information to observe whether or not the implementation is working and configured correctly.

We have to specify the correct levels for each log statement all over the code, and only use the Logger interface for any printouts. This effectively replaces all System.out.println calls. We currently have around 460 calls to the logger utility, which is our wrapper for log4j. The wrapper is created so that we have the possibility to change the entire log implementation without any major changes to the source code, we can also limit the penalty of log statements further than disabling it them from the configuration, by merely returning from the wrapper. It has not yet been required to use any of these techniques.

6.7.2 StatusFile

While the logging utility gives a continuous stream of events written to the log file, there is no easy way to fetch the state at each node from this log stream at a specific point in time. We require to have an easy to access and easy to parse, view of the state at each node at well defined point in time. State is information about, for example, the amount of information in the local store, knowledge with respect to metadata, communication parameters, number of messages sent and received. The purpose of the information is to get a quantified view of the nodes' parameters at this moment in time. The parameters must be extendable so that we can add more as the need for them arrives. The information is intended to be used for postanalysis.

The implementation of the StatusFile class was the answer to this need. It is inspired by the status files in Linux often found under /proc or /sys. /proc/meminfo is a good example, where one can find an updated total view of the memory usage. We then defined a class which runs as a daemon inside the middleware and at given time periods write all parameter to the specified file. The file is only updated if there is a change in its content. This gives the middleware less disk accesses since parameters tend to change only in short time periods or bursts. By placing a reference to an internal structure of some subcomponent, the StatusFile will be able to monitor this structure over time. This gives us a clean interface when using the StatusFile utility. In normal running of the component, no further action is required, other then registration of the internal structure. The StatusFile will constantly monitor the property and upon change write the new status to the status file.

At registration of the structure an identifier is specified, this will be the identifier that one can search for when parsing the status file at a later point. Currently only ArrayList type of structures are supported, this is chosen because almost all components use some sort of ArrayList for their internal structure. This can be extended in later versions to support other types of structures.

The monitoring and output can be of two types, either a list of the content, which will use the *toString()* function for each element or as a count of the elements inside. The latter is the most used and also the fastest. It is the most valuable with respect to graphs and statistics, were we tend to prefer a quantified number of the size instead of the actual content when analyzing.

6.8 Software testing

When developing a distributed application or middleware there are many things that are possible error sources. Many of these are bugs or badly developed code, or communication problems of sort. To debug and handle the problems that arise under execution of a protocol or a distributed application, one needs to first locate the error (what is going wrong), then track down its source (which node and in which stage), then try to correct the problem regardless of its nature, bug or not. If one could at an earlier stage verify to some extent that parts and functionality work correctly, one could focus the search into other parts of the system rather than do a system wide search to begin with. This would save time in having less bugs and smaller areas to search. This is one of the reasons for testing early in the development cycle.

There exist many paradigms and methods for how and when to test. They span from “test first” to “test to verify”, from functional to behaviour orientated testing. I will not go into the area of test methodology, but I have with great interest used the methods both as a help and as an exploration to see if they fit my work situation with great success.

My major concern is to give some confidence to the code produced, I wanted to see if the developed code was actually performing the task according to its spesification. The goal is to do this before deploying to the emulator in a full out test scenario or as early in the development iterations as possible. This way I know that an error probably has been introduced in the last iteration of functionality, while the previous have been tested. By doing this I also found that by testing my classes I could change the implementation via refactoring without the fear of breaking the behaviour of the system. This, regression testing, has been another important part of my motivation to keep, and especially maintain, the test in an automated form and over time. As the different subsystems have changed, I can monitor their effect on my system and see if my earlier functionality breaks. This effort has been substantial and one could argue that it might have been wasted time, but I am convinced that it has both reduced my development time by focusing the effort on the problem area, and that I have made less bugs while the tests have revealed them early in the process.

Below I describe the jUnit test framework and the technique of mock objects to isolate the code under test. Both have been essential parts of my test harness.

6.8.1 Unit testing

Unit testing is the method where one tests the smallest working units of code. In our case those are the developed classes. For example, the ChainManager which organizes the execution of each protocol, has unit tests testing its functionality, these are combined called a test suite. The tests can be quite basic, like after inserting a ChainElement into the manager, checking that it exists inside. To more complex tests where a full chain is set up of mocked ChainElements that finish without any aborts and checking that the running of the chain actually executes the correct end-function. Then, by changing one of the elements to be a ChainElement that always aborts, and then checking that the ChainManager gets the property isAborted and that it throws an Exception. As we see, there is an attempt to exhaustively check for any case that might break the defined functionality.

In addition to the verification of the classes can the testcases serve as a documentation of how the classes are used. By looking at all the tests the test case documentation emerges, this is good documentation for developers that come into a new project.

To automate this process both in development and execution, I have used a common framework called jUnit. It is a unit test framework for Java, with integration for eclipse. It gives a visual representation of executed, failed and completed tests, and makes it easy to make new tests. Our build and package supports jUnit in the way that it can run all of the test suites to check for failed tests for every build, packaging or deployment. This helps us to not introduce bugs or unwanted changes.

6.8.2 Mock objects

As mentioned before, the use of unit tests will require isolation of the unit under test. This implie that if there are dependences used during the test these dependences needs to be controlled so that they will not influence the test. To be able to control the dependencies, we use a technique called mock objects. What we actually do is to make a new implementation of the dependent class which does exactly what we want it to do. It can for example return a specific value or on each consecutive call to a function return objects from a predefined list. In this way we will be able to test only the one unit, while we have full control of its surroundings. By gaining this control we can both see how the unit reacts under normal condition, but we can additionally check who it reaches under non-normal conditions. We can introduce corner cases and other anomalies to see how the unit will be affected.

By using mock objects I have been able to verify the correctness of protocols and other network related features. Both the BroadcastResponder for topology monitoring and each of the steps in all protocols are tested using mock objects. These tests border over into regression testing as they do not test a confined small unit of code, but rather large stacks of functionality, this is needed to check that the interfaces can cooperate.

Chapter 7

Instrumentation and Test setup

This chapter describes how the implementation is tested by emulation. This testing differs from software testing since it is concerned with the entire middleware and not only parts of it, and by using an environment to instrument the middleware that resembles that of a real life scenario. I will describe the method of testing by emulation before describing the concepts and goals of each scenario, finishing off by describing in detail how each scenario is implemented.

The purpose of the metadata information sharing is to enable and improve the flow of information throughout the network. To be able to ensure that we actually reach this goal we need to test out different implementations and compare the actual performance. This is done on the basis of the absolute requirements and the performance requirements stated in Section 4.1. Through the process of testing, we will find areas where we can improve, this information can be fed back into the development. It will, if the results are positive, give more confidence to the architecture and design of the information synchronization mechanisms.

To be able to test the system, we need to construct scenarios that will expose the wanted behavior. For this purpose we construct scenarios consisting of the movement pattern for a set of node, along with the actions of each node at a given time. A scenario is like a time line, it starts at one point, instructs the nodes with the actions and movement, and terminates at a given time. The instructions to the nodes can be movements or actions. Movement can affect the nodes communication. Actions can insert data, allocate data and so on, which affect the behavior of the node. Through such actions and instructions we can construct a close to real life scenario that can be rerun multiple times. By doing this we can isolate situations that are interesting with respect to the specific part we want to test.

During a scenario each node logs the status and actions. This is done in both system and status logs. System logs are maintained for debugging and reflect system behavior, error and problems. The status logs reflect the information and knowledge that the node contains. These logs will be used for post processing. In post processing we will look for multiple attributes; efficiency, consistency and durability. The post processing is an analytical phase that will pull information

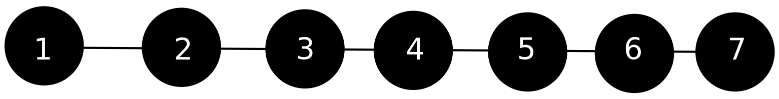


Figure 7.1: A schematic view of a chain scenario

from the logs and construct graphs and indicators reflecting the performance of the implementation. Measurements are represented either on a time line according to there time stamps, or as a single value. Based on the values we can draw conclusions on the performance. It is important to note that this holds if we change the implementation only. To ensure that the differences in test results are reflecting only changes in the implementations we are required to keep everything else static. This means that the scenario, actions and connectivity is equal for each of the test run, and that only the implementation is changed.

To test the information sharing in the ad-hoc environment we need to construct multiple scenarios that represent different cases for the network behavior. It is important to not only see how well the information disseminates in the optimistic cases, but also in the cases where the implementation struggles and delivers poor performance. We have constructed each scenario for one specific purpose.

7.1 Chain

The chain scenario is the simplest of the scenarios. The nodes form a chain, i.e. each node is the only path between its two neighbors, as viewed in Fig 7.1. This scenario is static, that is, it has no movement, thus the connectivity is the same from start to end. We use this to see how the basic features of the implementation are working. This is not a real-life scenario, but we will clearly see differences in how well each of the implementations work in its most basic from.

Regarding actions, the scenario will, after creation and stabilization, instruct node 1 to create a table replica, this will cause the creation of a metadata element, which will trigger synchronization. This will disseminate the element to the neighboring node 2, which triggers its synchronization and so on.

The goal of the scenario is to see if the implementation is able to get a 100% information sharing. This initial test will give us an indication of the basic performance, and if it does not pass, the other scenarios will not pass either.

7.2 Grid

While we in the chain had a maximum of 2 neighbors per node, this number is 4 in the grid. The grid is another controlled static scenario, which is made to see how the efficiency and dissemination attributes perform on a more complex structure, as in Fig 7.2. This scenario is more realistic than the chain scenario since there are more neighbors, and therefore more routes between two nodes. This means that optimalization give notably better performance.

The action performed in this scenario is creation of a new table. The action performed creates a table on node 1, this creates metadata that is going to

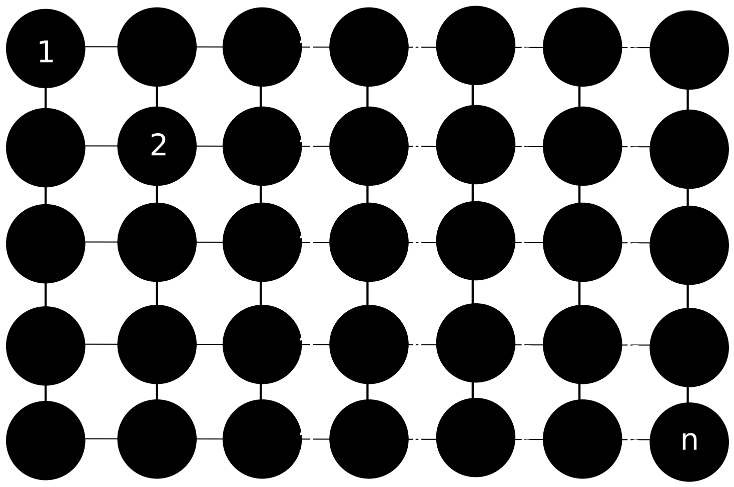


Figure 7.2: Simple schematic of a grid scenario

propagate throughout the grid, reaching neighboring nodes quite quickly. The more long term check is to see if it ever reaches node n. This is a test of the reliability of the protocol. We can also see the optimization of protocols that handle multiple neighbors more clearly.

Speed and total bits transferred will vary using different protocols, thus giving us an indication of the success.

7.3 Message ferry

The message ferry scenario is designed to show that one node can transfer metadata from one partition to another, as shown in Fig. 7.3. This scenario is semi-static, since only one node, C, is moving. Both partitions A and B are stationary and out of contact with each other.

The scenario starts off with C in contact with partition A. C gathers the metadata residing in A and starts its movement. At first it looses connections with A, it later gains contact with B and stops inside the B partition. Status recordings from the B partition will show how this ferry concept performs, in terms of metadata transport. Status of the metadata storage is recorded during the movement, to see the dissemination of elements.

The goal of the scenario is partly proof of concept, partly durability with respect to the delay tolerant dissemination.

7.4 Merge

As the scenarios become more complex, the results will become complex to analyze. It is vital to have dynamic scenarios to simulate more real-life scenarios. The merge scenario is constructed to stress the situations where we expect that the largest transfer loads will be registered. It simulates two partitions of nodes meeting, as shown in Fig. 7.4.

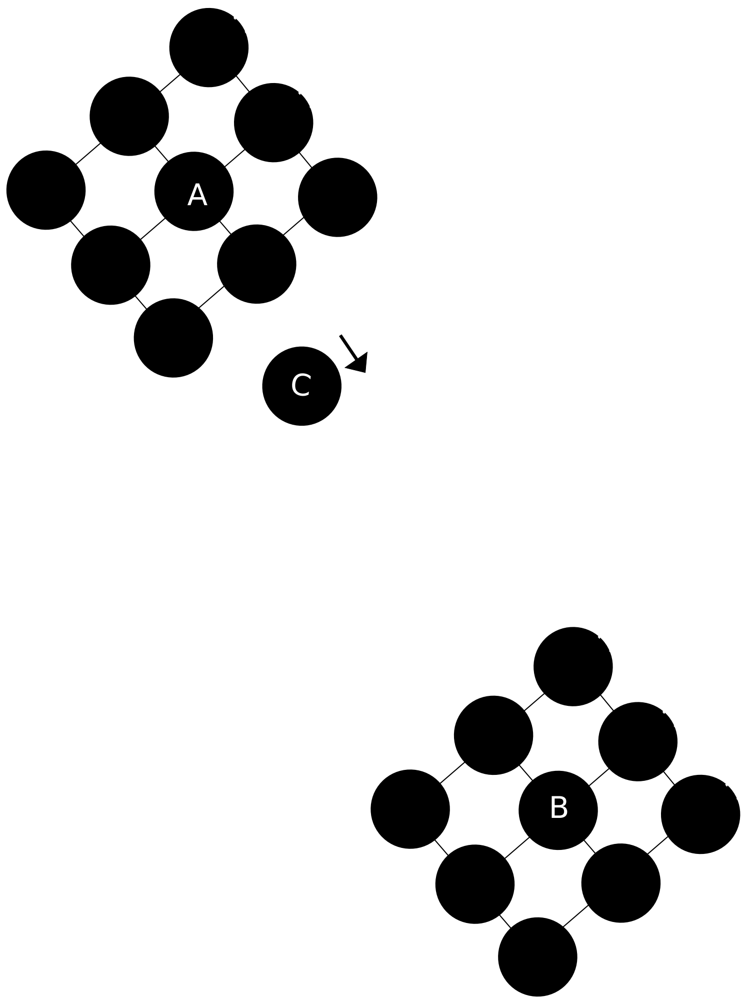


Figure 7.3: Schematic view of the ferry scenario, partition B and A including C, the ferry

Actions performed at the start of the scenario includes only creation of new table instances, after a given time period the nodes start moving against each other, until there is an overlap of the two partitions. Data about the status of the metadata storage is recorded during the movement and for a given time after the movement has ended. Ideally we will see a stabilization of synchronization protocols between the nodes, indicating a 100% dissemination. More specifically, a stop in network traffic because all nodes are equal.

The goal of the scenario is to identify how the different protocols handle massive trigger load, i.e. many new neighbors. It is also constructed to see how fast the protocols disseminate the information to all nodes, and if the protocol then stabilize and network load stops. Total network load is also important to differ between the implementations.

7.5 Test scenario implementation

I will now describe how the scenarios are made by describing the actual implemented scenarios in detail. These are the scenarios runned in the Test result chapter, and can be used as a reference to gain more insight into the results.

7.5.1 Chain scenario

The chain scenario consists of 10 nodes, as seen in Figure 7.5, configured to form a line of nodes. The scenario starts without any connectivity, after 1 second all consecutive nodes are connected, eg. 1 and 2, 2 and 3 and so on. Each node has then 1 or 2 neighbours and there exists only one partition. After 55 seconds, another table is created at node 1. The table will be disseminated throughout the network. The scenario shows if the hop by hop dissemination works, by looking at the graphs and verifying that all three protocols manage the hop by hop dissemination.

When using the semantic protocol I configure the group to include nodes 2 and 9. These are located in opposite corners of the scenario, 2 is between 1 and 3 and 9 is between 10 and 8.

7.5.2 Grid scenario

The grid scenario forms a grid construction of 20 nodes, as can be seen in Figure 7.6, in a pattern of 5 times 4 nodes. The distance between the nodes is such that each node only has connectivity with the node directly above and below in addition to left and right, which gives each node 4 neighbours. This does not hold for the outer nodes which have fewer neighbours.

At one second into the scenario, each of the nodes is connected to form the grid topology. 60 seconds into the scenario one new table is created on node 1. The groups when using the semantic protocol are formed of nodes 2 and 19, which are located on different sides of the scenario but are not the nodes that starts the dissemination.

7.5.3 Merge scenario

The merge scenario is started with two partitions, each of which is formed 1 second into the scenario. Each partition consist of 9 nodes, 18 in total, as seen

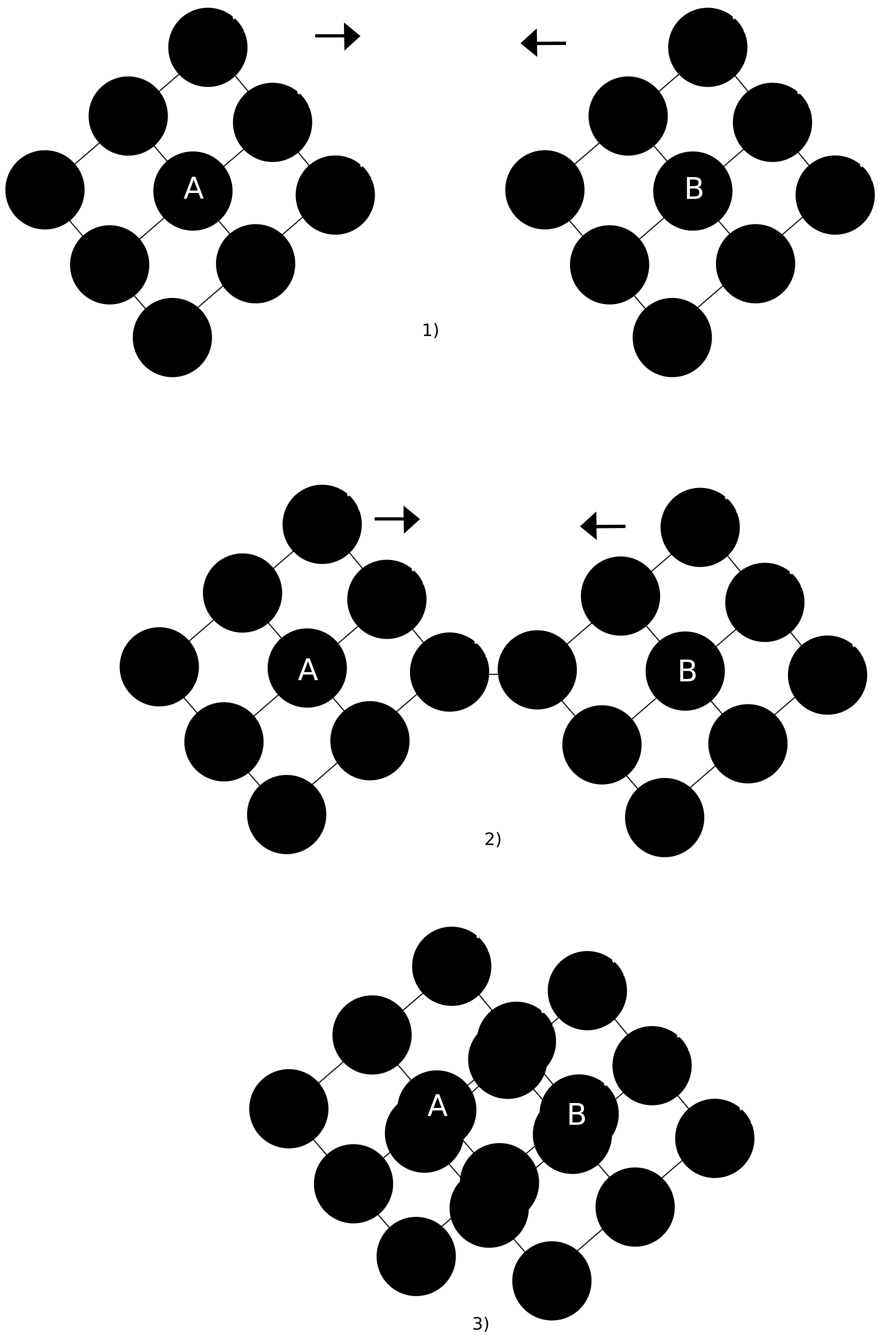


Figure 7.4: Schematic view of the merge of cluster A and B

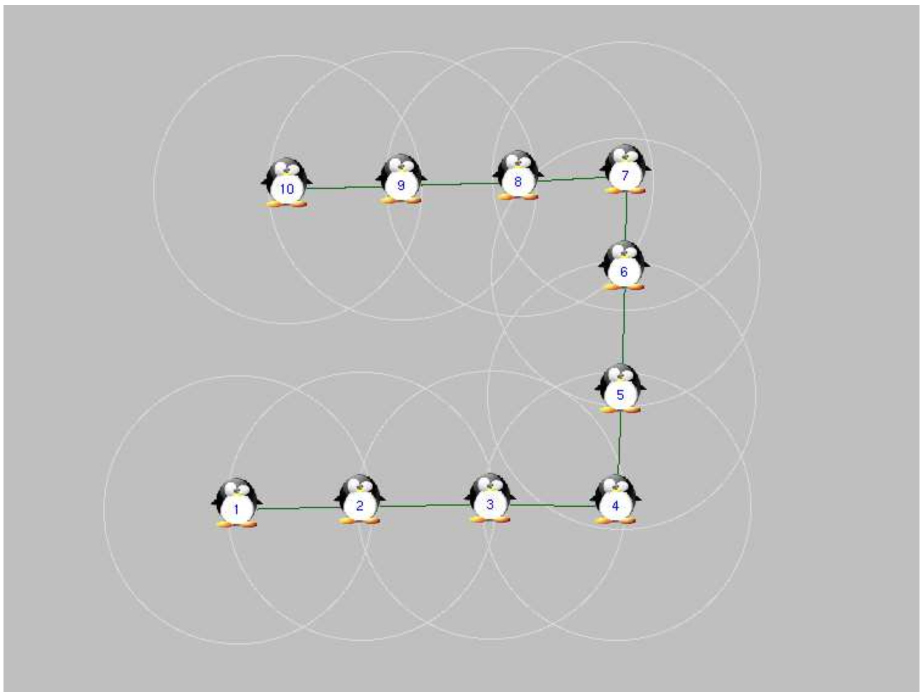


Figure 7.5: Screenshot from the emulator GUI, showing the chain scenario

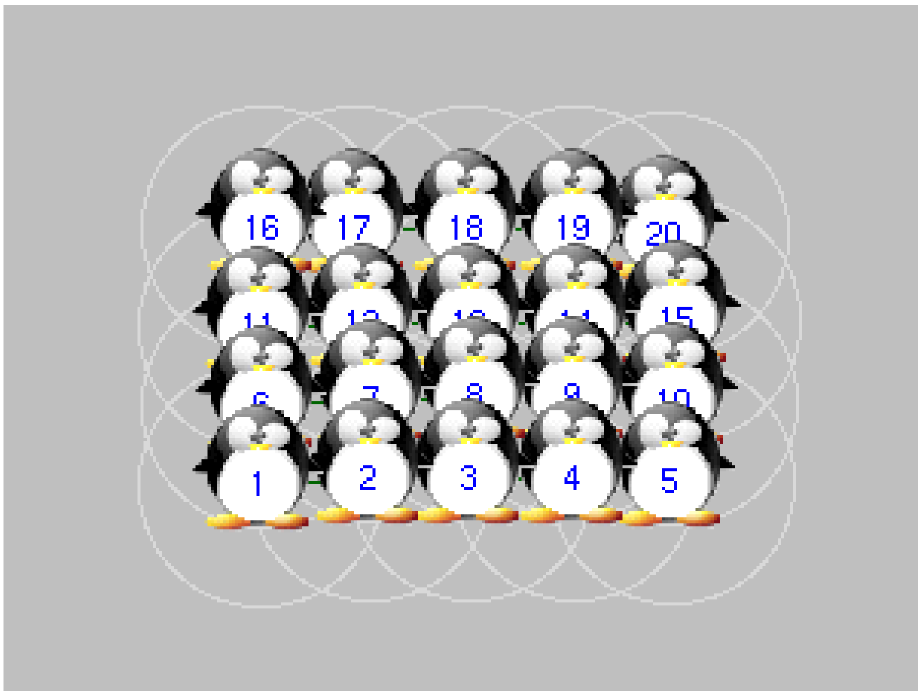


Figure 7.6: Screenshot from the emulator GUI, showing the grid scenario

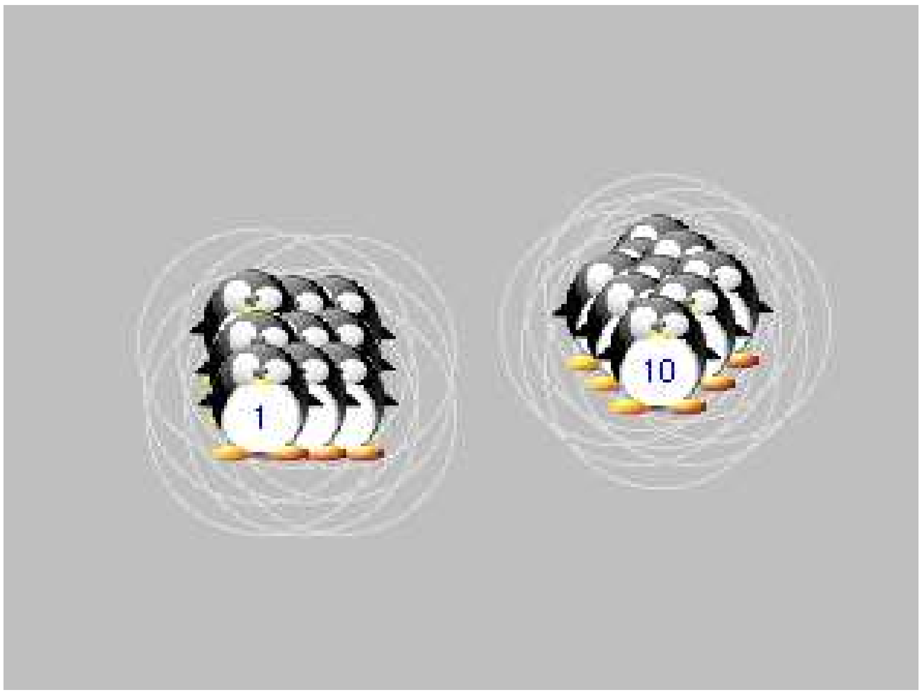


Figure 7.7: Screenshot from the emulator GUI, showing the merge scenario

in Figure 7.7. At 60 seconds the partitions will merge. This happens when the two first nodes in range will make connection; as the two partitions overlap, more nodes will get connectivity with the other partition. At 70 seconds both partition will overlap totally, which means that all nodes in the partitions have neighbours that belonged to the other partition.

No action in form of table creation is performed, only the change in connectivity. When considering the semantic scenario nodes, 1 and 10 are in the same group. They are in different initial partitions, and are not the ones that meet at the first encounter.

7.5.4 Ferry scenario

The ferry scenario consists of 17 nodes, at first separated into two partitions. One consists of 9 nodes, the other of 7 nodes plus the ferry. At 1 second into the scenario, both partitions are formed by turning on the network. The ferry starts immediately to move through the first partition. At 40 seconds into the scenario the ferry will lose connectivity with the first partition, heading for the second, this is where the screenshot in Figure 7.8 is taken. At 47 seconds, will the ferry connect with the second partition. Still moving through the partition, stopping at the end of the scenario inside the second partition on the opposite side of the entry point.

The semantic groups are formed by the ferry itself, node number 1, and a node inside the second partition, number 2. The group member in the second partition is located in the middle of the partition, hence is not the first encounter

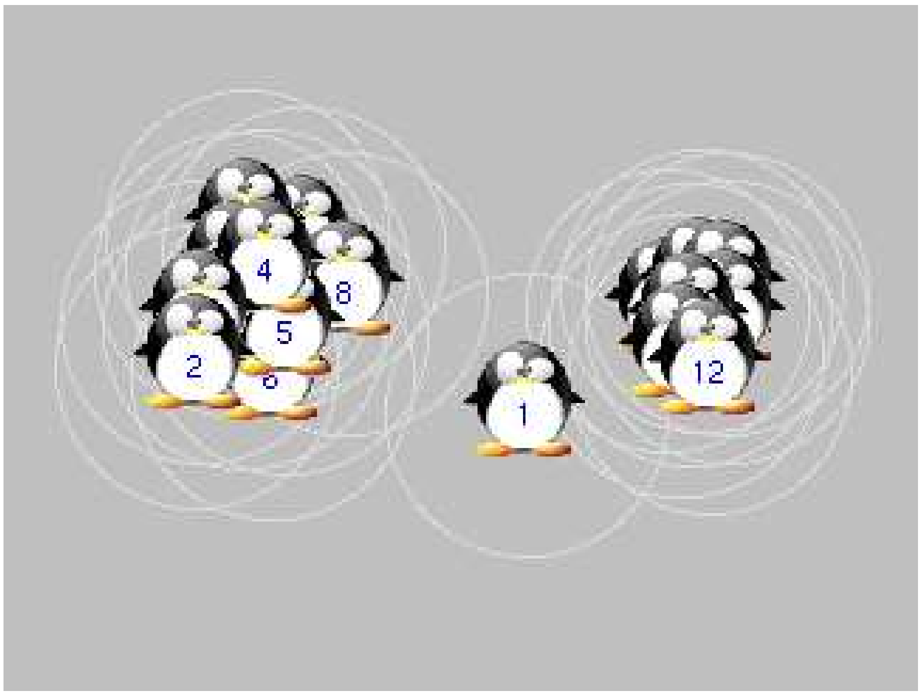


Figure 7.8: Screenshot from the emulator GUI, showing the ferry scenario

of the ferry.

7.6 Tools used and made

Few tools was used making scenario files prior to my contribution, random or hand written scenarios where used. I needed a more controlled environment where I could construct scenarios. This was accomplished by using the NAM editor, and finding a way to translate its output format to some thing that could be used by the NEMAN iemul GUI. With the pipeline described in this section I am able to construct and run a scenario of my specifications without hand coding scenarios.

7.6.1 NAM

NAM is a network animator [24], which can construct and view topologies. In addition to its own format it can save and view NS2 files. NAM is able to construct networks for the NS2 simulator, nodes in these networks can be mobile or stationary. Agents known from NS2 can be placed in the network to generate traffic. Other features available in NS2 are also available in NAM but not introduced here since they will not be used by the NEMAN emulator.

I have used the NAM editor to construct the NS2 files that are used in my test scenarios. It generates NS2 formatted files. The NS2 files are specifying the scenario and in the NAM editor, one can use a time line to specify movement of nodes. These scenario files can later be converted to NEMAN GUI readable files.

7.6.2 NAM format to NEMAN converter

There are certain format differences between the NAM written format and the format readable by the NAME GUI. These are;

- numbering, NEMAN scripts require 0-n, NAM writes 1-(n+1)
- variables in NEMAN scripts are used with an trailing _

When these changes are made by the *n*s2neman script will the *s*ensnet.pl script, from the NEMAN-1.1 util package, be used to construct the connectivity information needed by NEMAN GUI. The NEMAN GUI uses this information to instruct TOPOMAN about connectivity.

7.6.3 iemul

iemul is the GUI for NEMAN, it is part of the NEMAN-1.1 GUI package. It can read modified NS2 scenario files with the additional connectivity instructions made by the sensenet.pl script. It can also send control packages to TOPOMAN which is used in my thesis for instructing the nodes. These control messages need to be hand written into the scenario files.

iemul can play, pause and rewind the scenarios, it graphically shows the movement pattern and connectivity on the client machine. iemul will make the control packets and the instructions to NEMAN, these are sent as the scenario specifies.

7.7 Measurements and simulation system

To be able to perform the test cases described, we will need to use an emulator environment. This environment will enable repeatability and make running of the test cases easy and reliable. We are using NEMAN to perform the emulation, and since NEMAN uses NS2 format for its scenario files, we can take advantage of the message passing feature from NS2 scenario files.

The concept is rather simple, the scenario files are going to be the single source for test cases. This means that in addition to movement patterns and connectivity, they will also handle behavior. Behavior is all that the middleware is able to do, we use some simple behaviors called actions like; log status, create new table, create new metadata, and query for available metadata. Via this behavior we will be able to start the test case from the emulator interface, and each test can be run in exactly the same way each time.

To realize the concept of single source test case files, we need to get the messages sent from the emulator GUI into the emulator. Inside the emulator environment, messages will be routed to the correct node or nodes. Each node should be able to pick up a message and perform the wanted behavior. This outline a design shown in Fig 7.9.

The messages are sent from the emulator GUI together with the control messages to the emulator. At the emulator they are picked up by software

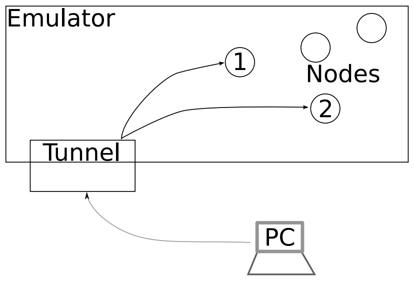


Figure 7.9: Simple outline of the simulator, with communication flow marked by arrows from the PC(emulator GUI) to the nodes inside the emulator

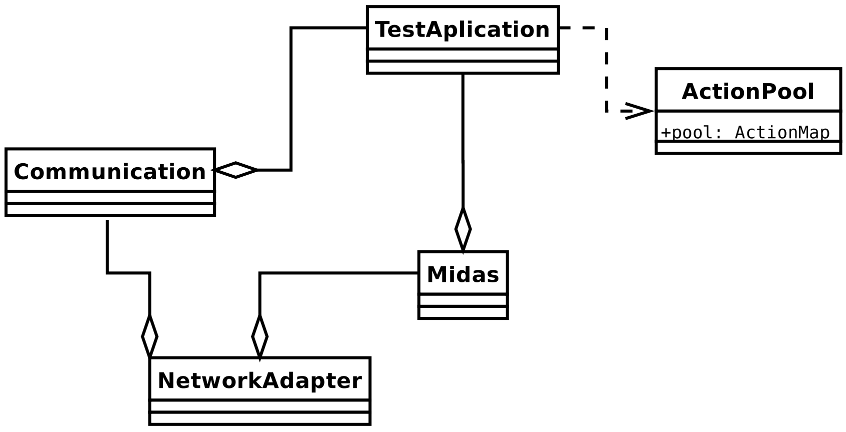


Figure 7.10: Simple outline of the test case application, that implements a remote control over the nodes

which tunnels the messages into the emulator to the IP address specified in the scenario files. At this point the nodes have to be ready to pick up the messages and interpret them.

7.7.1 Controlling nodes

To control the nodes I made an application that uses the MIDAS middleware. The application is controlled by the messages sent to the nodes. By doing the control this way I get to test that the middleware as a whole works in the proper way. In addition, the tests will be more precise to what they would be in real life situations. Since the middleware is used in more of the same way as a normal application would.

When designing the application two major requirements surfaced; One, the use of the middleware needed to be as if it was a user application using it. Two, the actions that a node could perform, had to be extendable. By accepting the requirements the design had to include a separate communication channel and a pool of actions that could be extended to new actions at a later point.

As we can see from the UML diagram overview in Fig 7.10, there is a separate communication unit for the application. This unit uses the same interface to the network layer as MIDAS does. By separating these two communication channels there will be no interference in MIDAS middleware by the control messages to/from the nodes. This will make both debugging easier and the results more reliable because the middleware is not affected by the control messages.

The test application is only executing actions from the action pool, this makes the test application extendable so that it can perform any task inserted into the ActionPool. The actions are performed in and application and will affect or operate the middleware through the MIDAS interface.

7.7.2 Logging and status

The logging of system data like size and count of knowledge base is done through a status file on the node. The file is a mirror of the status on the node at any given moment. The logging is done externally and according to a sampling rate. This is done through copying the file and noting the timestamp. In this way we can from the outside log the status without interfering with the application. How this is done is described in the implementation Section 6.7.

Chapter 8

Test results

The results from tests performed are laid out and discussed in this chapter. I start with the metrics used and the influencing factors. Next, I describe the evaluation techniques and presenting the measurements done during the test runs both in tables and graphs. Further the protocols are evaluated against each other based on performance in the same scenario and each protocol by is discussed, evaluating it over all scenarios. Finaly I Identifying thire successes and limitations and also evaluate thier scalability.

8.1 Metrics

I have evaluated each scenario using each of the protocols, measuring the meta-data storage and communication cost from each node. By doing this I am able to verify and check that each protocol fulfils the absolute requirements. In addition, I will look into how well they perform with respect to the performance requirements described in Section 1.3. All measurements are done in the context of time, which gives me the ability to see the dissemination as a function over time.

In this section I will describe the metrics used to measure and evaluate each protocol.

8.1.1 Sample interval

The tests are instrumented using the StatusFile described in Section 6.7.2, by reading the status files once each second, marking them with the current timestamp, and writing them to a new file in the logging directory. Latter this directory is packaged together with the tcpdump log, for transport and post analysis.

The script which copies the status files is quite simple. Each of the files are copied in sequence, which introduces a small delay between the first and last copy operation. After this is done a sleep command is invoked with the sample interval, one second in these tests.

8.1.2 Information storage count

The known information elements, in the form of metadata, are kept in the information storage. We use the concept information storage to denote storage of all data, local as well as shared in the local storage.

Each node has an initial information storage containing one metadata element, this is a table replica used by the DS component for versioning purposes. This versioning table is a normal shared table as any other table instance, but it is special since it is created at startup on all nodes. This will imply that in a partition of two nodes, each node will after synchronization have two elements in its information storage. These are its own versioning table and the other node's versioning table. The global information storage in this case consists of all its nodes' information storage, counting $2 + 2 = 4$. Since all MIDAS nodes have this feature of creating a table at startup, I can use this table as a measurement into how far dissemination has reached without the need for active allocation of tables by the test application.

To determine the absolute size of the information storage I will use the number of nodes, thus forming the information storage count (ISC), based on the fact that each node has one table replica at startup. From this number I am able to determine how many metadata elements each node has. The global ISC is well defined in all scenarios, by knowing the initial size of each ISC and the number of nodes in each partition. This is used to check if the absolute requirement is met, by asserting the measured global ISC to the calculated global ISC.

8.1.3 Dissemination time

I will measure the time used to disseminate from an action occurs until the ISC has reached the new total information level. By plotting the global ISC I can make graphs showing the time since scenario start, and the size of the global ISC by evaluating these graphs (see Fig 8.1), looking for areas where the global ISC goes from one known stable state to another. By finding the last seen stable measurement and the first new stable measurement, I can measure the time spent from synchronization start until end; this is the dissemination time for these elements.

In the static scenarios I have used one new table replica as the new information created. This information is created on a node defined by the scenario. Identification of the stable states is done by knowledge of how many nodes are in each partition, thus forming the ISC in that partition. Using the fact that each node has only one table present at startup I know that the ISC for each node should equal that of the number of nodes in that partition. To locate the next stable state I will look for the new wanted global ISC.

In the static scenarios the new ISC is one larger that the number of nodes in the partition. This represents that there is one table shared per node, plus one created by the scenario.

In the dynamic scenarios this metric is more complex. The new ISC is determined by the joint ISC of both partitions. To give a more practical example; In the ferry scenario one node goes from partition A to partition B. The initial ISC of partition B is determined by the number of nodes in the partition. After the ferry has moved, the wanted ISC is the initial ISC of partitions A and B

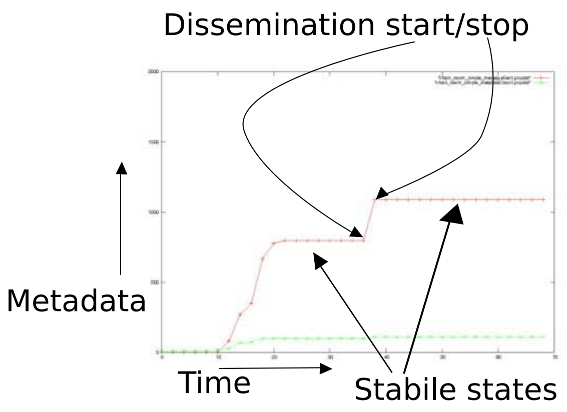


Figure 8.1: Graph showing the different stable states and start and stop of the dissemination

combined. Partition B should now be aware of all elements in B and all from A through the node mobility.

8.1.4 Message sent count

To disseminate information in the network, each protocol uses messages to communicate. This communication is represented in my metrics as the number of messages sent. Each message that is sent, contributes to each node's message counter. By measuring the number of messages sent, I can evaluate how much strain the protocols put on the network. By evaluating the message count together with the ISC I can analyze which protocols deliver high dissemination with a small amount of message use.

This metric does not give the finite answer to how large the network load is, because the message count does not include the size of the message. Further analysis and discussion of how good the indicators are, is done in Section 8.4.1. One can identify the activity level of the protocol by looking at the send rate. A zero send rate indicates no active protocols, an ever growing indicates a bug or error, a stable growing message count indicates activity. This indication is used to measure dissemination time, for this purpose is the message count well suited.

To take measurements of sent messages, I have instrumented each node to report its status to the status file regarding its message status. This is done in the MdsCrtAdapter which implies that we will not have any lower level packets included in the measurements. Additionally, this is only used by the MDS component, which gives us an accurate measurement of the MDS message count. For the duration of the tests presented in this thesis I will disable the DS component. This gives the message count more accuracy when studying only the GMDM component. This means that all messages reported sent, are from the GMDM component.

8.2 Influencing factors

In this section I will describe error sources and important aspects of the measurements. This is done to gain more clarity into what is presented, both with respect to what the numbers and graphs mean, and what they show or might not show.

8.2.1 Dissemination time inaccuracy

The dissemination time is based on the time differences from one stable state, until another stable state. The time between these two states is the dissemination time of the metadata disseminated. The action is, as described before, either a creation of metadata or a merge of two partitions.

To identify that a stable state is ending, I will find the last known stable measurement point. This point is found knowing the size of the ISC and the actions in the scenario. The first point in the next stable state marks the end of the dissemination.

While some of the protocols are fast in relation to the sample interval, some of the results might be too inaccurate. This is an effect of the instrumentation technique used, which takes one measurement each sample period. If a change happens between two measurements, this will first be recorded when the measurement is taken. Since the time span of some protocols is less than 3 seconds, this sample interval is too small to give an accurate measurement of the dissemination time.

When a stable state is ending, the recording of the increase in ISC will be done up to 1 second after it actually happened. This is also the case when the dissemination has reached everyone, which is marked by the last node getting the new level of ISC. The recording of this ISC can be up to 1 second delayed. This gives the dissemination time an inaccuracy of +- 2 seconds, one at the start of the dissemination and one at the end.

8.2.2 Message size omitted

In the graphs presented all measurements are in terms of messages sent or in the number of metadata elements. When using only message count as the definitive quantity, one must remember that the actual size sent over the network for two messages can differ by large factors. In Section 8.4.1, I will explore the actual differences between the message count and the bandwidth use. It seems that the message count is an ok indicator of how good the performance is. But there is inaccuracy when only using the message count, especially in the Merge scenario.

Environment	Protocol	Messages sent	Message recv	Sec merge 100%
Chain	Simple	29,1	25,3	3
	Broadcast	4,6	2,8	1
	Semantic	3,1	4,7	1
Grid	Simple	62,5	57,0	3
	Broadcast	3,1	6,2	2
	Semantic	3,7	6,8	3
Ferry	Simple	21,8	21,8	4
	Broadcast	3,8	17,7	2
	Semantic	6,6	12,1	4
Merge	Simple	48,1	48,1	7
	Broadcast	18,6	185,9	4
	Semantic	29,0	44,5	6

Table 8.1: Messages sent and received during the dissemination phase. Dissemination time, is measured in seconds

8.3 Evaluation technique

To evaluate these tests I will use the goals stated in Section 1.3 and the graphs and measurements in this chapter. By comparing each of the protocols against the goals for information sharing I hope to find answers to how well each protocol performs in each of the scenarios. I will also look into the relationship between the different protocols used on the same scenario to see if there are protocols that perform better under some scenarios than others. The goal is to see if there exist protocols that is well suited for all conditions, or if a universal protocol is not one of the tested, and if so what would be the causes for the different performance.

The assessments are going to be done through graphs for time series data like dissemination, and in tables or histograms for quantified data without the time dimension.

8.4 Conclusions drawn from the data

The parameters in Table 8.1 is as follows; Messages sent is the average number of sent messages through the middleware, per node. Messages received are the average number of messages received through the middleware per node. All messages addressed to one node, including the broadcast address, get counted. Messages are counted from the start of the dissemination until the nodes stop sending messages. The stop in message sending indicates that all nodes are equal and that all protocol instances are finished. “Sec merge 100%”, the duration of the dissemination, this is the time it takes from the protocols start, until the 100% is reached. This time period is shorter than the period that is used from dissemination start until message sending stops. Timestamps are done only on whole seconds, thus can the measurements only be given in whole seconds.

Environment	Protocol	Bytes sent
Chain	Simple	563727
	Broadcast	22115
	Semantic	30188
Grid	Simple	2693948
	Broadcast	53895
	Semantic	76641
Ferry	Simple	359102
	Broadcast	68983
	Semantic	71226
Merge	Simple	731260
	Broadcast	360067
	Semantic	494226

Table 8.2: MIDAS traffic measured during dissemination

8.4.1 Bytes transferred

To get more concrete results of what goes on at the network level, I have measured the bandwidth use in each of the tests. This is done through use of tcpdump, and post analysis of the dump files. The measurement is done from the starting point of the dissemination. This is identified through analysis of the tcpdump file by looking for the control packets from the emulator GUI, which indicates the start of the static scenarios, or in dynamic scenarios the first message sent between the two first encounter nodes. For example, in the ferry scenario, the encounter between the ferry and the second partition. The measurement is done by adding up all packet lengths sent to every MIDAS node on port 1113. This will include all messages send from one MIDAS node to another, excluding all routing daemon traffic and the control packets.

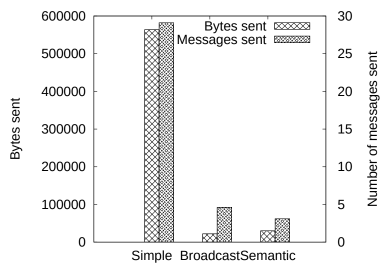
The measurement results are displayed in Table 8.2, and show each of the scenarios and protocols and the number of bytes sent in the dissemination period, described above.

8.4.2 Messages good indication of bytes transferred

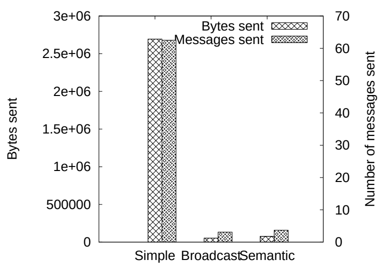
Message size can vary, this is not visible when only looking at the message count. I will asses if the trends are the same regarding bytes and messages, if this is the case will messages indicate, bytes usage and vice versa. For the trends to correlate will the increase in, for example, byte usage be proportional to the increase in messages.

As we can see from the plotted data in Figure 8.2 the byte count follows the message count to some extent. Some exceptions occur in the chain scenario (see Figure 8.2(a)) where the broadcast protocol sends more bytes than the semantic protocol, but sends fewer messages. This illustrates the problem with only counting messages and not bytes, since large messages are sent the comparison to the semantic protocol will be misleading. Also in the merge scenario we see that the broadcast and semantic protocols use almost equal amounts of bandwidth, even though the broadcast protocol reports less messages, which indicates larger messages.

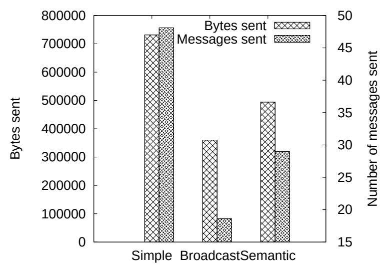
Even though byte count does not accurately follow the message count, we can



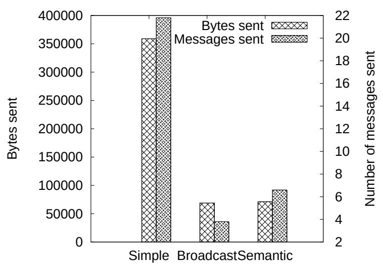
(a) Chain scenario



(b) Grid scenario



(c) Ferry scenario



(d) Merge scenario

Figure 8.2: Messages and bytes plotted side by side, to show the relation ship between messages and bandwidth usage

still use the message count as an indicator for bandwidth use. The reason for this is that the message count follows the actual bandwidth use to some extent, even though we would like a more close correlation. Messages used in the semantic protocol use a multihop routes which gives $packet\_size * hops = bandwidth\_use$, which is part of the deviation. What the message count still tells us is the active use of bandwidth, thus a reduction in the message count affects the byte count. I will use message count for absolute requirements, and to some extent to asses the performance requirements but use the bandwidth measurements when appropriate.

8.5 Dissemination results

In this section I will display the test results from the test scenarios. By doing so I will argument that we have fulfilled the absolute requirements and that some protocols perform very well with respect to speed, bytes and message count.

We will use graphs to display metadata counts over time. By analyzing the graphs I will find whether or not the protocol has fulfilled the absolute requirements. In addition I will identify the dissemination time between the two stable states. The protocols will be measured against each other within each of the scenarios. This will give us a clear view of the relative performance of the implementations.

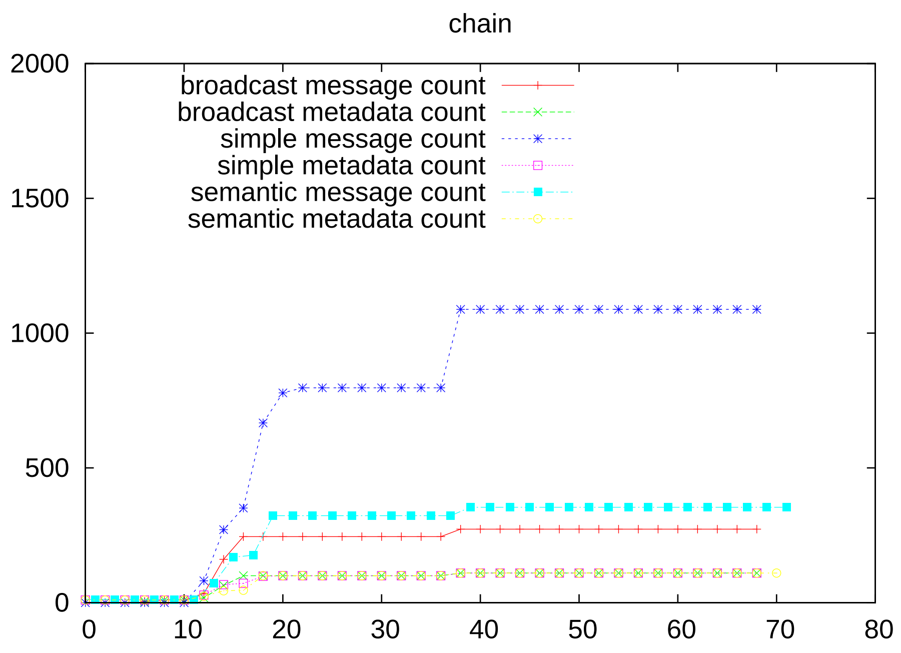


Figure 8.3: Plot using the total values for the entire network using the three different protocols in the chain scenario.

8.5.1 Chain configuration

Evaluation

When analyzing the chain scenario using the graph in Figure 8.3 and the numbers from Table 8.1 we discover some interesting results; small differences in time usage, very large differences in message count, and some unexpected results regarding the received message count.

Time differences used for dissemination in the chain scenario are almost none. The simple synchronization protocol uses 3 seconds and the two others use 1 second. Knowing that the sample period is one second and that the uncertainty is around +-2 seconds, these results have to be treated as more or less the same.

In the chain scenario, the simple protocol uses 9 times as much messages as the semantic protocol, and more than 6 times as much as the broadcast protocol. This is confirmed by looking at the bytes transferred, where simple synchronization uses 25 times as much bandwidth as the broadcast synchronization, and the simple synchronization uses 18 times more bandwidth than the semantic synchronization. The broadcast protocol is using more messages and less bandwidth than the semantic protocol, which indicates that it uses smaller messages.

To look at the overall scenario we evaluate the graph and find that the difference in total message count is 1:4 between the simple and broadcast protocols, this includes the setup phase. It is interesting to note that the overall difference is smaller than the studied dissemination period. The overall message use includes too many uncontrolled aspects to be analyzed in this context. I will therefore dissemination phase to measure performance.

Taking a closer look at the received messages (see Table 8.1) in the chain scenario, one notices that the broadcast protocol receives less messages than the semantic protocol. As the use of broadcast messages increase will the difference between received and sent messages increase, if the node that uses broadcast messages has more than 1 neighbor. Since a broadcast message is counted as 1 sent message and as n receives, were n is the number of neighbors. If no messages is lost to communication error, this will be the only way that the send count is larger than the receive count. In the case of the chain scenario, where the receive count is less than the send count, this has to be explained by transmission error. While no other phenomenon can explain the fact that a message is send with out any one receiving it, especially considering that these are static scenarios where all nodes has connectivity at all times.

When looking at the Table 8.1, we also notice that the broadcast protocol is sending more messages than the semantic protocol, which indicates that the semantic protocol outperforms the broadcast protocol, this is not true for the bytes transferred which is seen to be larger for the semantic synchronization. The explanation for this might be that the broadcast protocol sends more overview messages alone than as part of the semantic synchronization. The reason for this is that the multiple start points for the following broadcast synchronization is more efficient than the plain broadcast with respect to redundant overviews. We expected that the results would show that semantic would go faster but use more bandwidth. We have only shown that the semantic protocol uses more bandwidth, with no measurable speed gain. What we can see from the message count, is that the broadcast protocol shows signs of not being optimal with respect to messages versus information sharing in the chain scenario.

8.5.2 Grid configuration

Evaluation

In the scenario shown in the graph in Figure 8.4 and in Table 8.1, we see that the simple synchronization is struggling with the grid scenario. By looking at the messages used to disseminate one element, it uses 20 times as many messages as the broadcast protocol. While the semantic and broadcast protocols are more or less the same when looking at there message count. When looking at the bytes transferred a some what different view emerges, the simple protocol uses almost 50 times as much bytes to disseminate one element as the broadcast protocol, for the semantic protocol the relation ship to simple protocol is 35 times. This however does not show as clearly in the timing results, which can be related to the use of NEMAN (Section 2.7.2) and the timing uncertainty. As NEMAN gives each node very large bandwidth, this reduces the problems imposed by high bandwidth usage. This is most likely the reason for the small differences in timing.

The large differences in message count might be caused by how the protocol handles many neighbors. We see, from the scenario description in Section 7.2, that the connectivity is high. An average node has 3.2 neighbors. When a new metadata element is created, the simple protocol will need 4 protocol instances, while the broadcast protocol needs one overview to reach all its neighbors. If some of the nodes already are up-to-date on information this will lead to addi-

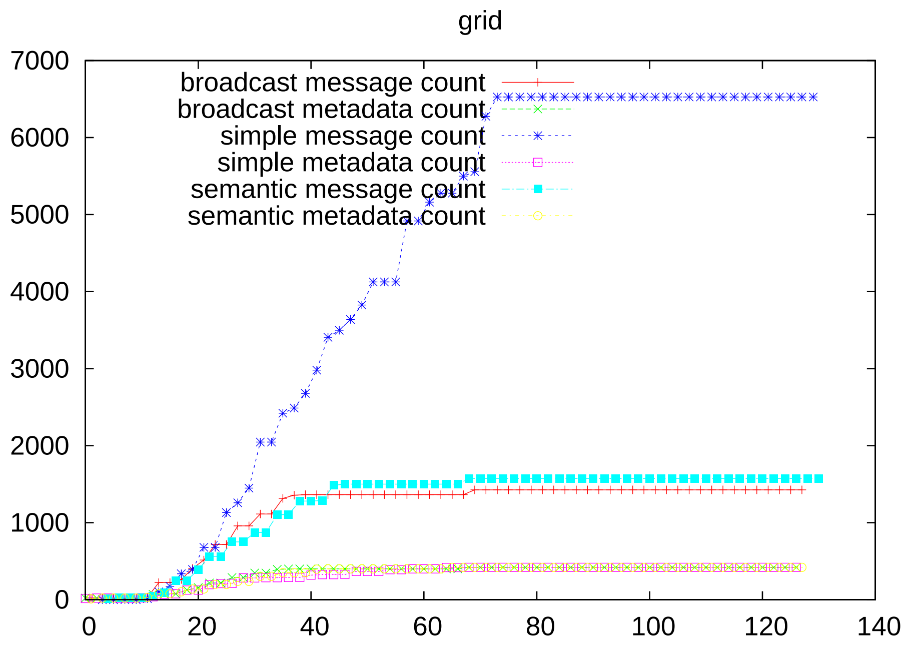


Figure 8.4: Plot using the total values for the entire network in the grid scenario, using the three different protocols.

tional savings. By analyzing the grid scenario we clearly see the consequences of this feature. The broadcast protocol outperforms the simple protocol by 20 times, regarding message use and 50 times regarding bandwidth. This result is higher than expected, and further discussed in Section 8.6.

We also see that the messages received and sent are more or less equal for the semantic and broadcast protocol, we can actually see a slightly larger message use by the semantical protocol. We were expecting good results from the semantical synchronization technique especially from the grid scenario. Because the recursive dissemination would require substantial iterations before reaching all the nodes. The increased number of start points was expected to be of great advantage for the semantical protocol. It might be that the scenario is still too small and the distance between group members is not large enough to give any substantial differences. Had both these interventions been cleared and the transfer medium reassembled more wifi networks than it does on NEMAN, we would perhaps have seen more beneficial results regarding the semantical protocol. There are too many uncertainties to speculate into the result of such an experiment.

There can be seen a pattern that the broadcast synchronization protocol is as good in the chain and grid scenario, but the simple protocol is sending more messages as the connectivity (neighbors per node) increases. This is an important aspect of the performance especially in relation to scalability, which will be discussed further in Section 8.6. The view is backed up by the bandwidth use, where the broadcast protocol increases its bandwidth usage from the chain scenario to the grid scenario by a factor of 2, the semantic protocol by 2.5,

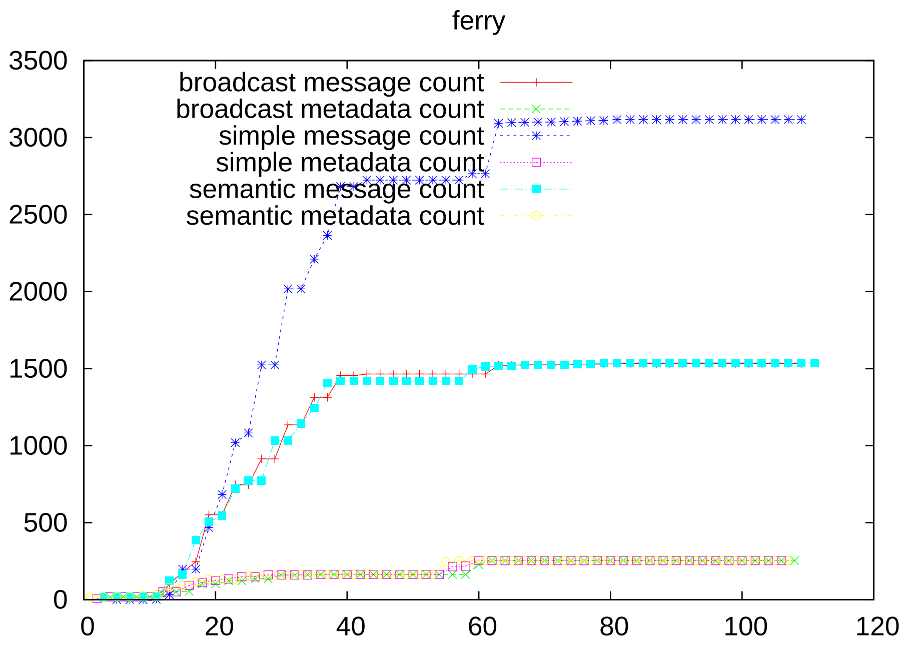


Figure 8.5: Plot using the total values for the entire network in the ferry scenario, using the three different protocols.

and the simple protocol by 4.7. These numbers show a relation between the protocols, but can not be related directly to scalability since there are to many differences between the chain and grid scenario. To measure scalability would more controlled and specific scenarios be constructed, that explored the neighbor ratio differences and the number of nodes with respect to scalability.

8.5.3 Ferry scenario

Evaluation

When evaluating the results shown in the graph in Figure 8.5, the ferry scenario, one must remember that this is actually a three part scenario;

1. initialization in two partitions of 7 and 10 nodes
2. disconnection from the smallest group by the ferry
3. the ferry enters the larger partition

We will look at the entering into the larger partition, as the durability evaluation, evaluating that the larger partition eventually consisting of 11 nodes, has the total information per node at 17. This shows that the protocol fulfils the absolute requirements also when subject to delay tolerant paths. By looking at the graph in Figure 8.5 we can confirm that the global ISC is equal to; $(6 * 7) + (11 * 17) = 229$ Where; 6*7 is the ISC for the smaller partition at the end of the scenario. 11*17 is the larger partition ISC at the end of the scenario. 229 is the global ISC.

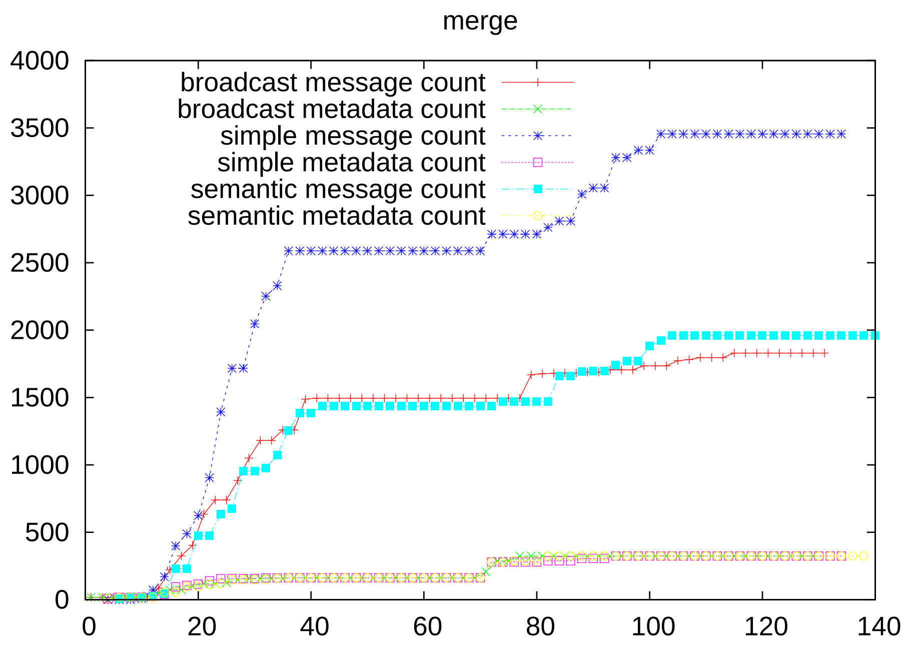


Figure 8.6: Plot using the total values for the entire network in the merge scenario, using the three different protocols.

Assessing performance, by looking at the different protocols operating in the same environment we see in the graph in Figure 8.5 and Table 8.1 that the differences are much smaller than in previous scenarios. As low as 1:3, between simple to semantic protocol, and the difference between simple to broadcast protocol is 1:5, regarding message count. The byte count tells us that the simple protocol uses 5.2 times more bandwidth than the broadcast protocol, and 5.0 times more than the semantic. This indicates that the ferry scenario does not differ as much between the protocols as the other scenarios. When we look at the time usage we see that the difference is still small, broadcast is still fastest, with the two others a bit behind.

It is, however, a 1:2 difference in messages sent when assessing broadcast against semantic. This difference in count is not seen in the other scenarios. However, the byte counts are almost the same differing only 3%. This indicates that the semantic protocol uses messages that are smaller in size than the broadcast protocol, this means less information per connection, which can lead to more overhead.

8.5.4 Merge scenario

Evaluation

The merge scenario is the scenario with the highest neighbor ratio, it includes connection changes during the actual merge which will stress the triggers on each of the involved nodes. This is reflected in the results by having many new protocols instances created, due to the *onNewneighbor()* trigger, and *newMeta-*

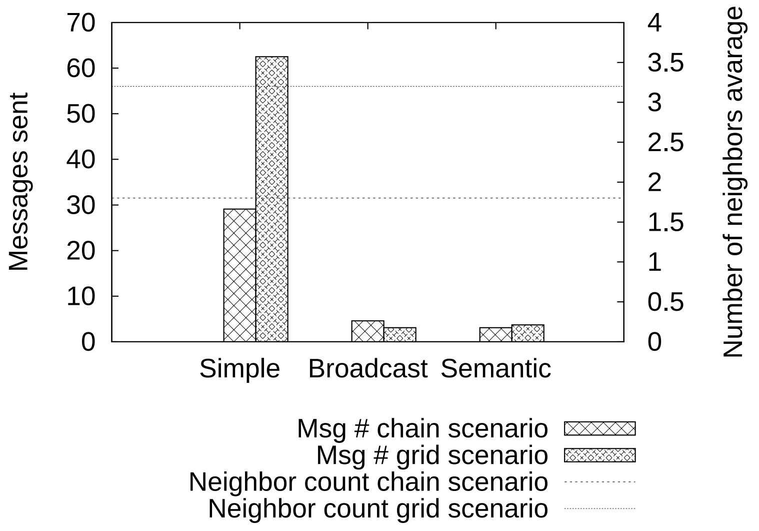


Figure 8.7: Performance plotted against neighbor count

data() trigger, which both trigger a synchronization protocol instance.

We can conclude from the Table 8.1 and by looking at the graph in Figure 8.6 that the broadcast is still fastest with 4 seconds, followed by the simple and semantic protocols on a shared second place around 6-7 seconds. When looking at the message count number, we see that the simple protocol is sending by far the most messages. Broadcast is sending least and semantic a place in between. The ratio between broadcast and simple is 1:2.6 and semantic vs. simple is 1:1,6. With respect to bandwidth is the ratio 2 between simple and broadcast, between simple and semantic the ratio is 1.5. Which states the performance gain of the broadcast protocol in the merge scenario.

What is very interesting is the receive count of the broadcast protocol, which is 10 times the number of send messages. This is a direct effect of the high neighbor count, since one node with 10 one-hop neighbors broadcast one message and gets heard by 11 nodes, by all neighbors and the sender itself. One would expect this to give efficient and very low bandwidth dissemination, however, the differences are not larger than in other scenarios in fact they are less. The high neighbor ratio would also give multiple dissemination paths, leading to multiple redundant overview messages that where to favour the optional responce broadcast protocol. This is neither seen as clear as we might expect. The reason for this might be the extensive trigger load, while the triggers get invoked very often will all protocol implementations need to respond with another instance of a synchronization. If this is the major cause to network load in this scenario, will the optimization done in the cutoff techniques might be as visible.

8.6 Performance versus neighbor count

To evaluate the protocols for scalability we have tried to find a correlation between neighbor count and bandwidth use, shown in Figure 8.7.

As the number of nodes in a partition increases, the number of neighbors will also increase, if the partition keeps its original physical size. If the number of neighbors increases, the protocols will be tested for scalability. As the number of neighbors is large, the protocols will be required to handle more nodes on each trigger, which can result in more bandwidth use. How well the increasing neighbor count is handled will show us how the protocol scale in dens networks.

I have used the results from the chain and grid scenario to show the scaling of the protocols. The results are shown in Figure 8.7. The graph shows messages sent using each of the three protocols, with the message count on the left hand side. The number of neighbors in average for each of the two scenarios is plotted as horizontal lines. The chain scenario has 1.8 neighbors per node, the grid has 3.2. The increase in neighbor count is substantial and can give us an indication on the scaling of the protocols. However these two tests are not enough to conclude on something definitive, and more tests and analyses must be performed to be able to give a definitive answer to how the protocols actually scale.

From these limited test results we can anyhow see some indications into how the different protocols react to the increase in node density. We observe that the simple protocol more than doubles its message use, while the broadcast protocol reduces its message use as the neighbor count gets higher. This indicates that the bandwidth use of the simple protocol dramatically increases with the neighbor count, which indicates bad scalability. While the broadcast protocol decreases its message use, as the neighbor count doubles, which indicates good scalability. When looking at the semantic protocol, we can see that the grid scenario uses a bit more messages then the chain scenario. This can be explained by looking at the two protocols that the semantic protocol consists of, which will explain the message use that increases. The increase over the broadcast protocol is due to the group synchronization phase.

However the picture drawn from the messages is not entirely correct. If we see the bytes transferred in relation to the scaling another picture emerges, which is different but still supports the view of scalability. The simple protocol goes from 533 727 bytes in the chain to 2 693 948 bytes which is 4,8 times larger, at the same time the broadcast protocol goes from 22 115 bytes to 53 895 byte the equivalent of 2,5 times. This still indicates the difference in scalability in favour of the broadcast protocol, but the difference is not as large as when we only look at the message count.

The subject of scalability must be looked into further to give a definitive answer. More tests with more nodes have to be performed and the correlation between neighboring nodes and dissemination patterns has to be analyzed to see if the patterns of dissemination changes as the protocol or topology changes. There must also be done an analysis of the problems with respect to the broadcast storm problem, which can possibly improve the performance of the protocols further.

8.7 Performance results

8.7.1 Simple protocol

The simple protocol was implemented according to the epidemic routing principle. No optimization is done to enhance its performance. When we evaluate the protocol over the different scenarios we can observe that also this simple implementation fulfils the absolute requirements, even though the 100% dissemination takes some time, all nodes get to know all information elements in all scenarios.

It suffers from lack off no cut-off techniques and use of the one-to-one synchronization method. This is slow and bandwidth costly, since every neighbor will be contacted directly. This is costly since the listner node also needs to perform the same tasks in the opposite direction due to the trigger *onNewMetadata()* (see Section 5.3). These two inefficient techniques degrade the performance of the global dissemination.

The decision to not do optimization in combination with the one-to-one technique gives the protocol a bandwidth use that is magnitudes larger than the other protocols. The bandwidth use is so large that it would probably inflict with normal operations in the ad-hoc network.

As the simple protocol is only a first implementation, there is no expectations into how the protocol is to perform. Anyhow, the bandwidth use of the simple protocol is a magnitude of 25 time more bytes transferred in the chain scenario, of 50 in the grid scenario, of 5 in the ferry scenario and of 2 in the merge scenario, than the bandwidth used by the broadcast protocol.

8.7.2 Broadcast protocol

The Broadcast protocol is optimized for performance and lower bandwidth usage. If we measure the protocol against the requirements from Section 1.3 we identify that the protocol fulfils the absolute requirements. Regarding the performance requirements the broadcast protocol performs well with respect to dissemination speed and bandwidth use. The results from each of the scenarios, looking at the graphs, achieve a 100% dissemination and in the fastest time for all scenarios. As we look at the number of messages used, we can observe that the use of bandwidth is lower than both the simple protocol and semantic protocol. Compared with the simple protocol the broadcast protocol uses from 2,5 to 20 times as few messages, and uses from 2, in the merge scenario, to 50 times less bandwidth than the simple synchronization, in the chain scenario.

The optimization done in the broadcast protocol clearly gives better performance and higher dissemination speed than the simple implementation. The less use of bandwidth is also an important optimization that favours the broadcast approach.

8.7.3 Semantic protocol

The goal of the semantic protocol was to speed up the dissemination process with focus on getting the information to a set of group members. As we see from the graphs the group synchronization is slowing down the global dissemination process compared to the Broadcast protocol, this is due to the overhead of multihop synchronization and delay for doing the group synchronization first. I expected that the overall dissemination would speed up as an effect of the overlay network that the group was forming, this has not been the case in the scenarios tested. I assume that the advantage of overlay network and multiple start points is going to prove themselves on larger scale topologies, especially where communication is more unreliable or slower than in NEMAN. But we have not seen this effect in any of the scenarios performed in this thesis.

The additional overhead is stated to be assessed against the overall gain of the semantical protocol. Both measurements of message count and bytes transfer show that the semantic protocol introduces more overhead over the broadcast protocol. This overhead ranges from no difference in messages sent in the static scenarios, up to 30-40% more messages sent in the merge and ferry scenarios, compared to the broadcast protocol. When we look at the bytes transferred we clearly see the actual overhead affecting bandwidth ranging from 3% in the ferry scenario to 36% in chain, 37% in the merge and up to 42% in the grid scenario more bandwidth used than the broadcast protocol.

Group dissemination speed

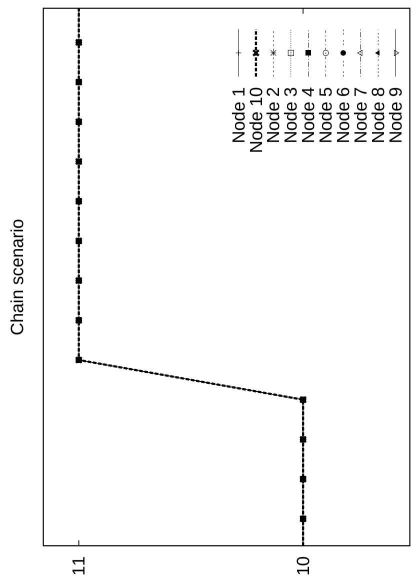
The semantic protocol has two levels of synchronization that are used in sequence, as described before. I will now assess the inter group dissemination speed. This is the output of the first synchronization protocol. Since the results from the scenarios have not proven the semantic protocol to be as fast as expected, we will try to see if the inter group synchronization is at all faster than the global dissemination, with respect to the group members.

By plotting all nodes on the same time line along the y axis and their ISC along the x axis we will be able to see how the group members which should receive metadata from the group, are performing in relation to the other nodes. To more easily see how the group members in question perform, the plot for this node is in bold. The graphs can be viewed in Figure 8.8.

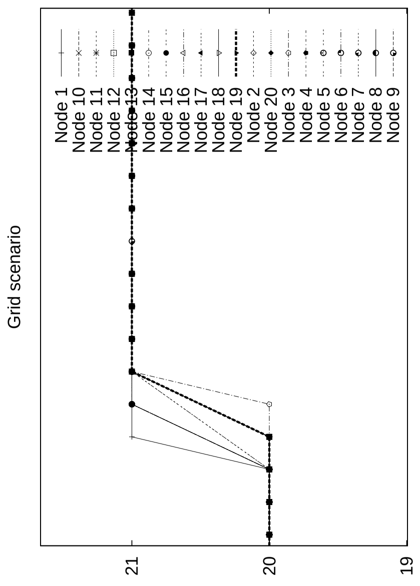
Chain scenario (Figure 8.8(a)) does not show any interesting results, all nodes are moving from knowing 10 metadata elements to 11 in the same sample. No difference can be established.

Grid scenario (Figure 8.8(b)) is a bit more interesting, it shows a late start for the group member synchronization and that the receiving member is getting the element as one of the last. The group synchronization is not speeding up its receiving rate. This can be explained by looking at when the first group member receives the metadata, in fact it can look as if nodes number 19 and 2 get the metadata at the same time. The receiving of metadata to the group is reliant on the dissemination of metadata from the source, node 1, to one of the two group nodes, 19 or 2, this explains the delay. However, by observing that the two group members receive the metadata at the same time, we can say that the group synchronization functions correctly.

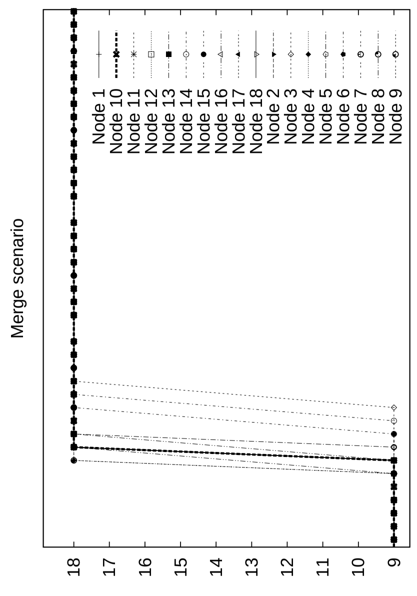
Merge scenario (Figure 8.8(c)) gives us another example of the group node not being the quickest. One needs to remember that the two group members are not the first to merge. Again we can see that nodes number 1 and 10, which are in different partitions, move from 9 metadata elements to 18 at the same sampling. This indicates that the propagation from group member to group member goes fast.



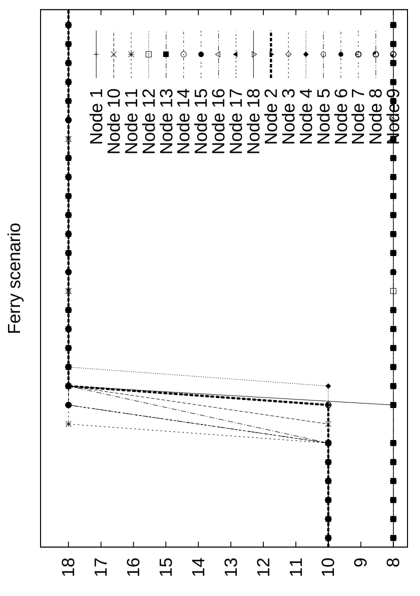
(a) Chain scenario, group consists of node 1 and 10



(b) Grid scenario, group consist of node 2 and 19



(c) Merge scenario, group consists of node 1 and 10



(d) Ferry scenario, group consist of node 1 and 2

Figure 8.8: The metadata plotted on a per node basis, representing the receiving node using a bold line

Ferry scenario (Figure 8.8(d)) where the ferry is part of the group gives us a clear example on what we have been seeing in the other scenarios. The group member enters another partition, invocation of the triggers happen, and the group member is updated. This can be seen while the ferry, the only one going from 8 metadata elements to 18, reaches the 18 metadata level at the same time as the receiving group node. An interesting aspect is that the only source of new metadata is the ferry, so the nodes that reach the new level of metadata, 18 elements, received these from the ferry. Some of the nodes do this before the ferry reaches the new level, indicating the delay of the semantic protocol during the merge.

Group synchronization evaluation

By studying the graphs in Figure 8.8, we see that the group synchronization works, and that it is fast among the group. It also reveals that the protocol makes the group nodes less responsive to synchronizations from other nodes while they do the group synchronization. This was clearly visible in the ferry scenario, where the single source of new metadata did not apparently store the new metadata elements, even though multiple other nodes had gotten their elements. This is a delay which can be introduced by processing power shortage or other non-protocol-related causes.

To more clearly see the advantage of using the synchronization protocol, tests were one of the group members are the source of the metadata. In such a scenario would the semantic protocol be more suted and possably show more efficientency, by doing so can more clear conclution be drawn on the performance gains over the broadcast protocol.

We can identify that the wanted behaviour from the semantic protocol is correct, since the group is propagating the elements fetched by one, to all others. This further strengthens the theory that the causes for the bad results in my scenarios are due to small scenarios or the fast emulator. It can also be due to the fact that the scenarios are constructed so that they reflect badly upon the semantic protocol, by giving the metadata other propagation routes than over the group nodes. This can be seen in the grid scenario, where multiple nodes, which where located more than one hop away, has received metadata before the group member, which was located only one hop away from the origin.

This leaves the semantic protocol with only increased overhead, and no real speed gain. Not due to the protocol but rather to the fast broadcast protocol, that over these short distances, and in this environment, outperforms the semantic protocol.

Chapter 9

Conclusion

We set out to handle information sharing in rescue scenarios using ad-hoc networks. Since information sharing is done alongside other applications also requiring network bandwidth, there are strict requirements into how the protocols are to perform with respect to both bandwidth and response time. By using metadata we can disseminate small information elements to all nodes in the network. To limit bandwidth usage we have implemented an approach that uses the neighbor relations, propagating information from node to node in a recursive manner. To optimize the node to node propagation, extensive testing and analysis have been performed. This has resulted in protocols spanning from the basic simple protocol to the more advanced broadcast and semantic protocols. Optimalization are done based on knowledge of the physical characteristics of the wifi and ad-hoc environment. We have shown that our information sharing protocols provide a global view of the information in the network. We have further shown that delay tolerant features are supported, and work as expected.

The protocols were developed in an incremental process starting with a simple implementation inspired by the epidemic routing protocol. The development focuses on durability, so that an element created in the network will be disseminated to all. The protocol has a simple outline using a one to one, recursive synchronization technique to propagate metadata to its neighbors. As the strengths and weaknesses of the protocol emerges, a second protocol was developed, taking advantage of the radio medium, using broadcast messages, hence the name broadcast protocol. It uses a one to many synchronization technique, reducing the overhead and also reducing the number of multiple dissemination paths, by not forcing already synchronized nodes to perform another synchronization. By using this optimalization the performance increases dramatically. Another attempt to further enhance the performance was done with the semantic protocol, inspired by using overlay networks as used in publish subscriber systems like DENS, the protocol uses the notion of groups among the nodes by giving the members of its own group first priority, thus forming the overlay network, and dissemination to these members is done faster. As a side effect we expect faster global dissemination by using this technique under the impression that multiple starting points were going to, disseminate the network faster. However there were not found measurable differences in dissemination time to the entire network, only to the group nodes.

The requirements put upon the protocols are twofold, absolute requirements that set the functional requirements, and the performance requirements that give us a metric for making optimalization. The absolute requirements include;

- 100% dissemination. All nodes network are required to know of all meta-data elements.
- Maintaining a global view. This requires that from every node all meta-data will be visible.

And the performance requirements;

- Low bandwidth.
- Dissemination speed.
- Search response time.

By using these metrics the protocols have been evaluated revealing the differences. I performed a set of test scenarios, exposing the protocols to different - more or less - realistic behaviour and movement amongst the nodes. By post analysis of the measured test data the absolute requirements were found to be valid, and the performance requirements evaluated. The absolute requirements hold for all implementations over all scenarios.

The broadcast protocol outperformed the simple protocol by magnitudes both in respect to its message use and bandwidth, also regarding dissemination speed was the broadcast protocol faster. This clearly shows that by optimalization the epidemic routing principle for information sharing in ad-hoc networks, we can reduce the bandwidth and time use by large quantities.

In analysis the concept of the semantic protocol looks promising, however we have not gathered data that shows that the semantic protocol is faster than the broadcast protocol, with respect to global dissemination. It introduces an overhead both in bandwidth use and a time delay. I have argued that the semantic protocol would perform better in relation to the broadcast protocol if the bandwidth in the emulator NEMAN were closer to that of the real life and if our scenarios had been bigger or more suited to fit the semantic protocol.

The performance is measured against the requirements, to find the bandwidth usage and dissemination speed. The search time is instant and does not vary between the implementations while the 100% dissemination requirements are followed. Our measurements have confirmed that the simple implementation can be outperformed by combining routing principles and the understanding of wireless environments. This is done through the broadcast protocol, which has proven to be an overall better utilization of bandwidth. We see that the optimalization saves up to 50 times the bandwidth when using the broadcast protocol vs. the simple protocol. The optimalization shows promising results when looking at the scenarios tested, as the chain versus the grid scenario, where the broadcast protocol uses 25 less bandwidth in the chain scenario and 50 time less in the grid scenario as the average neighbor count doubles. This indicates that the broadcast protocol can handle larger and more dense topologies. The Broadcast protocol gives the information sharing better responsiveness and at the same time uses less bandwidth. This successfully shows that some of the concepts of routing in ad-hoc networks are applicable to information sharing. As done with the broadcast protocol, one can build upon the research field of routing when developing protocols for information sharing.

During the development of the three protocols there was done analysis of the performance, which was fed back into the development. By doing so would the protocols, especially the broadcast and semantic protocol, perform better both with respect to dissemination time and bandwidth usage. This analysis in the development cycle was done by using the scenarios and the emulator, and by maintaining the test suite could I experiment with changes, without introducing bugs. This has lead to the one to many technique and other optimalization used in the implementations. The repeatability made the effects of the changes visible while the test cases did ensure that no unexpected behaviour was introduced. This gave freedom to experiment and confidence that no wrong functional behaviour was introduced.

The automation of scenario creation did early on show that new scenarios could be constructed in a matter of minutes not hours, as the case was before. This gives a more flexible way of emulating the behaviour and testing the actual performance of the implementation in a corner case.

During the development process there has been developed a framework for fast development of application using the ad-hoc network, also referred to as the MIDAS middleware. The framework consists of APIs ready to perform many of the tasks required by an application. Thus using the ready made framework will application developers in the future be able to use more time on the applications than the ad-hoc environment problems. As part of this thesis, a middleware implementation enabling application to send, and route messages, explore the topology, and store shareable data, has been developed. This is accomplished by developing a series of components handling, point to point message transport, integration against routing tables and sharing of data amongst nodes. This has resulted in a framework that have given support for both applications development, as well as internal protocols for information sharing.

By easy creation of the test scenarios, has testing of specific scenarios become easier. This is accomplished by using the NAM editor and a developed converter to the NEMAN format. When the scenario is constructed the accuracy of running the scenarios become more reliant while all nodes are controlled by a central unit, which can repeat the scenario multiple times. Before the development of the RollePlayer and the mechanisms of controlpackets, would only movement and connectivity be controlled by the control unit, with my contribution can this control now be extended to the behaviour of each node.

The MDS framework, which is the implementation product of this thesis, is extendable and usable by application developers also out side the MIDAS project. The frame work can be used in general to realize other distributed application using ad-hoc networks. The components can also be used separately, to take advantage implemented features, in addition to the use of the middleware as a whole..

9.1 Further work

As I have studied routing protocols during this thesis has multiple other techniques emerged. Only some of which are implemented and tested as part of this work. Further implementation of different techniques will probably give more efficient implementations.

The subject of limiting broadcast storms is highly usable for even the already implemented protocols. This will reduce the redundant paths of dissemination, and result in saving in bandwidth and dissemination time.

Probabilistic routing is another protocol, based on the same concept as epidemic routing but with an introduced probability. I have argued that this concept of probability in propagating the metadata elements can give more efficient protocols, this concept is also known from reduction techniques for broadcast storms.

The semantic protocol has shown to be giving a group high priority, however our test has not revealed any performance gains in the global dissemination. There needs to be done more tests in larger and on more realistic emulators to show if this concept is performing better than the standard nongroup approach. Also scenarios more tailored to the semantic protocol runned in on NEMAN should give interesting results. If so, dynamic creation of the groups would be interesting to give a more efficient selection of the group members with respect to the global dissemination. By selecting the nodes that are in key positions in the topology, the overlay network would always be optimal to serve the global dissemination. The overlay network can be constructed upon calculation done over the topology known from the routing daemon. Which positions in the topology that are key to efficiency when doing dissemination, has to be studied. This topology aware semantic synchronization is done both by the DENS overlay network, by distributed selection of the partition representative, and mentioned in the broadcast storm paper, when it selects its gateways and heads.

As specified in the performance requirements both the search time and 100% dissemination are important aspects of the information sharing concept. However, these requirements will require that all metadata is located on all nodes. The bandwidth used to disseminate the metadata to all nodes is substantial, to lower this bandwidth usage the requirements have to change. We could see a shift so that the search time could be "as low as possible" at the same time as an optimal dissemination rate is found. The wanted effect is that not all nodes are required to have all metadata if the search time is low. The bandwidth savings will be substantial if the proportional searches for metadata is lower than the creation of metadata.

References

[1] "Apache log4j 1.2 - log4j 1.2," Apache. [Online]. Available: http://logging.apache.org/log4j/1.2/index.html

[2] T. H. Clausen, *Optimized Link State Routing Protocol (OLSR)*, The Internet Society, Network Working Group, the official RFC for olsr, rfc3626.

[3] A. Datta, *Autonomous Gossiping: A Self-Organizing Epidemic Algorithm for Selective Information Dissemination in Wireless Mobile Ad-Hoc Networks*, 2004. [Online]. Available: http://www.springerlink.com/content/rt1jlbvd1x98xj74

[4] O. V. Drugan, T. Plagemann, and E. Munthe-Kaas, "Predicting time intervals forresource availability in manets," in *SUTC '06: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing - Vol 2 - Workshops.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 32–37.

[5] R. A. Elmasri and S. B. Navathe, *Fundamentals of Database Systems, fifth edition*, M. Goldstein, Ed. Addison-Wesley Longman Publishing Co., Inc., 2007.

[6] L. M. Feeney, "A taxonomy for routing protocols in mobile ad hoc networks, Tech. Rep. T99–07, 1, 1999. [Online]. Available: citeseer.ist.psu.edu/feeney99taxonomy.html

[7] E. Freeman, E. Freeman, B. Bates, and K. Sierra, *Head First Design Patterns.* O'Reilly, October 2004.

[8] K. A. Harras, *Delay Tolerant Mobile Networks (DTMNs): Controlled Flooding in Sparse Mobile Networks*, 2005. [Online]. Available: http://www.springerlink.com/content/pgeg51q68nj3fyhe

[9] J. Hassan and S. Jha, "Optimising expanding ring search for multi-hop wireless networks," *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, vol. 2, pp. 1061–1065 Vol.2, 12 2004.

[10] ——, "Performance analysis of expanding ring search for multi-hop wireless networks," *Vehicu Technology Conference, 2004. VTC2004-Fall. 2004 IEEE 60th*, vol. 5, pp. 3615–3619 Vol. 5, 26-29 Sept. 2004.

[11] "Part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications," IEEE, 1997, the 802.11 standard.

[12] A. Lindgren, A. Doria, and O. Scheln, "Probabilistic routing in intermittently connected networks," 2003. [Online]. Available: citeseer.ist.psu.edu/lindgren03probabilistic.html

[13] P. M. and P. T., "Neman: a network emulator for mobile ad-hoc networks," 6 2005.

[14] "Design: Middleware for connectivity and information sharing, design of the midas data space," MIDAS, 1 2007. [Online]. Available: https://project.sintef.no/eRoomReq/Files/ecy/MIDAS/0_37a6c/D2%5B1%5D.1_AnnexII_postValladolid.doc

[15] "Midas offical website," MIDAS, 9 2007. [Online]. Available: http://www.ist-midas.org/

[16] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu, "The broadcast storm problem in a mobile ad hoc network," in *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. New York, NY, USA: ACM, 1999, pp. 151–162.

[17] olsr.org, 3 2008. [Online]. Available: http://www.olsr.org/

[18] C. E. Perkins, *Ad hoc On-Demand Distance Vector (AODV) Routing*, The Internet Society, Network Working Group, the official RFC for aodv, rfc3561.

[19] T. Plagemann, K. S. Skjelsvik, M. Puzar, A. Johannessen, O. Drugan, V. Goebel, and E. Munthe-Kaas, "Cross-layer overlay synchronization in sparse manets," in *Proceedings of the 5th International ISCRAM Conference - Washington, DC, USA*, May 2008.

[20] N. Sanderson, *Metadata Management for Ad-Hoc InfoWare - A Rescue and Emergency Use Case for Mobile Ad-Hoc Scenarios*. Springer Berlin / Heidelberg, 11 2005, vol. 3761/2005.

[21] K. Skjelsvik, "Distributed event notification for mobile ad hoc networks," *Distributed Systems Online, IEEE , vol.5, no.8, pp. 2-2*, 08 2004.

[22] M. M. B. Tariq, M. Ammar, and E. Zegura, "Message ferry route design for sparse ad hoc networks with mobile nodes," in *MobiHoc '06: Proceedings of the seventh ACM international symposium on Mobile ad hoc networking and computing*. New York, NY, USA: ACM Press, 2006, pp. 37–48.

[23] "Ns2 documentation," UC Berkeley, LBL, USC/ISI, and Xerox PARC, 10 2007. [Online]. Available: http://www.isi.edu/nsnam/ns/doc/

[24] "Nam: Network animator," University of Southern California's Information Sciences Institute (ISI), 4 2008. [Online]. Available: http://www.isi.edu/nsnam/nam/

[25] A. Vahdat, "Epidemic routing for partially-connected ad hoc networks," *Duke Technical Report CS-2000-06*, 2000.

[26] H. Young-Bae, Ko.; Nitin, "Location aided routing in mobile ad hoc networks," *Wireless Networks*, vol. 6, no. 4, pp. 307–321, Sep 2000. [Online]. Available: http://www.springerlink.com/content/t074q3x1j7972u52

[27] W. Zhao, M. Ammar, and E. Zegura, "A message ferrying approach for data delivery in sparse mobile ad hoc networks," in *MobiHoc '04: Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing*. New York, NY, USA: ACM Press, 2004, pp. 187–198.

Appendix A

Cd content

This chapter describes the content of the cd. It contains the scripts used to set up the test enviroment when running my experiments. In addition is all source code added both for the MIDAS project and the implementation of the DENS protocol.

A.1 /implementasjon

This folder contains both the MIDAS implementation and the DENS implementation.

A.1.1 /implementasjon/midas

What lies in the *midas* catalog is the MDS implementation as described in this thesis. The MDS implementation has been developed further after this branch has been taken out, which was done the 31. january 2008, some modifications is done by me with respect to this thesis. Includeing removal of the DS hook in GMDM to only have the GMDM protocols runing. It is built with maven which gives the catalog structure and the utility used to build it. To build the entire code,

1. install and configure maven, can be found on http://maven.apache.org/
2. run the *install.sh* script in the deps catalog.
3. run *mvn eclipse:eclipse* to generate project files for the eclipse ide.
4. run *mvn test* to run the entire test suite.
5. run *men assembly:assembly* to generate the jar file used on the nodes. The jar file is build and packaged in the *target* folder.

A.1.2 /implementasjon/infoware-dens

The DENS implementation is also built with maven, follow the above instructions to install and build. It requires Java 6.

A.2 /test-setup

The test-setup catalog is all the scripts used to run the tests. The route catalog contains the following scripts;

- *read.* Made to read the status file once a second
- *run.sh.* Starts one node on the given tap, with the given parameters to the VM.
- *run.nodes.sh.* Starts multiple nodes from first parametert to last, third parameter is used to spesify protocol. *./run.nodes.sh 1 20 broadcast* starts the broadcast protocol on tap 1 to 20 in giving the nodes correct IP addresses
- *run_tests.sh* Takes the following parameters "number of nodes" "protocol" "name of test", and runns all tests checking that the olsr enviroment is ready, loging all status files and tcpdump of *tap0* (getting all communication in the emulator) and upon ending will the entire folder, named according to the ¡name of test¿, be made into a tarball ready to be downloaded for post prosessing.
- *tunnel.pl* tunneling the controlpackets from the GUI to the nodes

A.2.1 /test-setup/scenarios

The scenarios are located in this catalog. They are ready to run in the iemul GUI.

A.3 /script

This catalog contains all scripts used to parse status files into gnuplot files preserving all parameters. Also scripts for generating the graphs displayed in the thesis from the gnuplot files are contained in this catalog. Scripts are done in both ruby and bash script.

Appendix B

Source code

In this appendix I list some of the source code that might be of interest without opening the entire source code.

B.1 Chain manager

```java
package midas.mw.mds.gmdm.chainManager;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import midas.mw.mds.logging.MdsLogger;


/**
 * Chaining manager. Planed to be used in protocol development
 *
 * Tought to be used in this way
 *
 * - over load the manager with another class which does the adds
     of the elements in the constructor
 * - the new class is calling the runChain
 * - the new class is listening on boutght end and aborted and does
      the actions required when each of them is called
 *
 * - the configuration array is only for internal use, and every
     new class extending from this must make its own protocol
 * - the map configuration is intended as inter step communication
     and service locator
 *
 * @author aslak
 *
 */
public class ChainManager {

  ChainManagerEndInterface end = null;
  int currentPoint = 0;
  private List chainElements = null;


  public ChainManager()
  {
    chainElements = new ArrayList();
  }

  /**
   * Runing a chan of elements
   * @param configuration A map of configuration, any class which
       extends this must
   * have there own protocl on how this is populated
   * @throws ChainException
   */
  public void runChain(Map configuration)
    throws ChainException
  {

      for(int i = 0; i < chainElements.size(); i++)
      {
        currentPoint = i;
        try {
          ((ChainElement)chainElements.get(i)).run(configuration);
        }catch(ChainAbortedException e)
        {
          MdsLogger.info("Chain stoped due to an abortion;" + e.
              getMessage(), this);
          return;
        }catch (ChainException e) {
          throw new ChainException("Trouble with the "+ i +" chain
              element", e);
        }

      }

      end(configuration);

  }

  /**
   * for setting up the chain, one can add a chain element, it will
        be added to the end of the chain
   * @param element the element to add
   * @throws ChainException
   */
  public void add(ChainElement element)
    throws ChainException
  {
    chainElements.add(element);
  }

  /**
   * for getting the current chainelement
   * @return the current chian element
   * @throws ChainException
   */
  public ChainElement getCurrentPoint()
    throws ChainException
  {
    if(currentPoint > chainElements.size())
      throw new ChainException("Current point not active or added",
          null);
    return (ChainElement) chainElements.get(currentPoint);
  }

```

```java
    /**
     * getting the starting point
     * @return the starting point
     * @throws ChainException
     */
    public ChainElement getStartPoint()
      throws ChainException
    {
      if(chainElements.size() == 0)
          throw new ChainException("No chain elements added", null);

      return (ChainElement) chainElements.get(currentPoint);
    }

    /**
     * For when ever a chain is aborted
     *
     * @param message a explainatory message
     * @param cause if a exception is the reason, else null
     * @throws ChainException
     */
    public void abort(String message, Exception cause)
      throws ChainException
    {
      throw new ChainException(message, cause);
    }

    /**
     * setting the end action
     * @param end
     */
    public void setEnd(ChainManagerEndInterface end)
    {
      this.end = end;
    }

    /**
     * Function which is run when ever all the elements are done
         succsesfully
     * @param configuration the configuration as a servicelocator for
         alle the elements
     * @throws ChainException
     */
    public void end(Map configuration)
      throws ChainException
    {
      if(end == null)
        throw new ChainException("End listener not specified", null);
      else
        end.end(configuration);

    }



}
```

Listing B.1: The ChainManager is made to ensure sequential execution of the steps.

```java
package midas.mw.mds.gmdm.chainManager;
```

```

import java.util.Map;



abstract public class ChainElement {


  /**
   * function to start this chain element, called when it is this
       elements turn
   * @param configuration configuration passed on from previous
       chain element
   * @throws ChainException
   */
  public void run(Map configuration)
    throws ChainException, ChainAbortedException
  {
    throw new ChainException("not implemented element", null);
  }


  /**
   * called when a element wants to abort the hole prosess
   * this gives every element a veto
   * @throws ChainAbortedException on throwing this, the manager is
        to stop the hole chain and call its own abort method
   */
  public void abort(String reason, Exception cause)
    throws ChainAbortedException
  {
    throw new ChainAbortedException(reason, cause);
  }
}
```

Listing B.2: The ChainElement is the abstract class which must be subclassed for a element, such as a protocol step, to be used in the ChainManager.

B.2 Request handler

I will now list the RequestHandler, and the SyncManagerController which is one of the components using the RequestHandler extensively.

```
package midas.mw.mds;

import java.util.Vector;

import midas.mw.mds.internalinterfaces.MdsInternalInterface;
import midas.mw.mds.internalinterfaces.MidasMessage;
import midas.mw.mds.logging.MdsLogger;

/**
 * Class for easy use of message passing
 *
 * intended use:
 *
 * construct new object, --> constructor is called and initiated
 * start the thread, by invoking .start() --> run is called and
     done, from here on the handler lives until it is deregistered
     from the waitingfor response usually done at the end or some
     abort method
 *
 * protocol is an extended class of request handler
 *
 * protocol = new ProtocolController(mds); //starts up the handler
 *
 * protocol.onReceivedMessage(message); // passes a new message to
     the handler
 * protocol.start(); //starts the protocol which runs upon the
     handler
 *
 * when a new messages arrives to the handler
 * protocol.onReceivedMessage(message); // passing on the message
     to the handler will invoke the run method
 *
 * @author aslak
 *
 */
public abstract class RequestHandler extends Thread implements
    MdsCrtCallBackAdapter {

  protected String requestId = "";
  protected String toComponent = "";
  private Vector messages = null;
  protected MdsInternalInterface mds;
  private boolean registered = false;

  /**
   * Used by extended classes to implement other then standard
       patterns of use.
   *
   * WARN: when used the user class is responsible for the proper
       setup of the handler and for any errors. for normal use use
       other constructor
   * the propertys is
   * <ul>
   *  <li>requestId</li>
   *  <li>toComponent</li>
   *  <li>registration</li>
   * </ul>
   * @param mds
   */
  public RequestHandler(MdsInternalInterface mds)
  {
    this.mds = mds;
    messages = new Vector();
  }

  /**
   * Constructor intended to use locally, i.e. when a request is
       started at this node
   * @param mds Pointer to the running MDS component
   * @param requestId one can define the requestId, it has to be
       uniqe inside this subcomponent
   * @param toComponent the component to sendt to,
   */
  public RequestHandler(MdsInternalInterface mds, String requestId,
      String toComponent) {
    //TODO remove tocompoennt
    super();

    this.mds = mds;
    this.toComponent = toComponent;
    this.requestId = requestId;
    messages = new Vector();

    //registrating the requesthandler at the mds, which is routing
    mds.waitingForResponse(this);
    this.registered = true;
  }

  /**
   * Send the message to the receiver specified in the message.
   *
   * The request ID is appended to the message here, hence this do
       not needed to be done before
   *
   * @param message
   */
  public void send(AbstractMdsMessage message)
  {
    message.setRequestId(requestId);
    //MdsLogger.debug("sending message "+ message.getClass().
        getName()+ ", requId: " + message.getRequestId(), this);
    mds.getCrt().send(message);
  }

  /**
   * Blocking receive, it returns when a message with the right
       request id and destination component is received
   *
   * @return The message received
   * @throws MdsTimeoutException Thrown upon a timeout, currently a
        fixed value of 10 sec
   * @throws MdsClientException If the MDS Client in form of the
       Request handler is having problems
   */
  public synchronized MidasMessage receive(int timeoutSeconds)
    throws MdsTimeoutException, MdsClientException
  {
    long startTime = System.currentTimeMillis();
    while(messages.size() == 0)
    {
      try {
        //the time out
        wait(timeoutSeconds * 1000);
        //checking if this is the timout or just a bogus notify
        //TODO sanity check
        if((startTime +(timeoutSeconds*1000)) <= System.
            currentTimeMillis())
          throw new MdsTimeoutException("Timeout occured "+
              timeoutSeconds + " sec", null);
      } catch (InterruptedException e) {
        throw new MdsClientException("Unasked for interupt!", null)
            ;
      }
    }


    /*
     * Remove System.out.println()... but this message is maybe not
         so important anyway
     System.out.println("+++ reading receive list");
     */
    if(messages.size() == 0)
      throw new MdsClientException("Read before update on receve
          message", null);

    MidasMessage retur =  (MidasMessage)messages.get(0);
    messages.remove(0);
    return retur;
  }

  public synchronized MidasMessage receive() throws
      MdsTimeoutException, MdsClientException {
    return this.receive(10);
  }

  /**
   * Start the thread
   */
  public abstract void run();


  /**
   * Method called by the routing authority. This message is
       delivered to
   * the receive queue. If the receive is currently blocking, it
       will return after the receive
   * buffer is updated
   */
  public synchronized void onReceivedMessage(MidasMessage message)
      {
    messages.add(message);
    MdsLogger.debug("[" +this.hashCode() + "] received a message "
        + message.getClass().getName() + ", " + message.
        getRequestId(), this);
    notifyAll();
  }

  /**
   * The component this hander is destined to
   * TODO remove
   * @return
   */
  public String getToComponent()
  {
    return toComponent;
  }


  /**
   * Get the request id of this requesthandler
   * @return
   */
  public String getRequestId()
  {
    return requestId;
  }

  /**
   * Set the request handler
   * use on response to a request
   * @param requestId
   */
  public void setRequestId(String requestId) {
    deregister();
    this.requestId = requestId;

    register();
  }

  /**
   * Register the handler
   */
  public void register()
  {
    mds.waitingForResponse(this);
    registered = true;
  }

  /**
   * Deregister handler from queue
   */
  public void deregister() {
    this.registered = false;
    mds.notWaitingAnyMore(this);
  }

  /**
   * Check whether handler is still registered
   * @return true if registered, false if not
   */
  public boolean isRegistered() {
    return registered;
  }

}
```

Listing B.3: The RequestHandler is made to make the middleware developer's life easier by transparently useing the asynchronous network as a synchronous one. In addition the RequestHandler will provide a session concept for the application to use.

```
package midas.mw.mds.gmdm.SyncManager;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Vector;

import midas.mw.mds.AbstractMdsMessage;
import midas.mw.mds.MdsTimeoutException;
import midas.mw.mds.RequestHandler;
import midas.mw.mds.ds.MdsDs;
import midas.mw.mds.gmdm.GmdmMessage;
import midas.mw.mds.gmdm.GmdmSynchronize;
import midas.mw.mds.gmdm.chainManager.ChainAbortedException;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.gmdm.chainManager.ChainManager;
import midas.mw.mds.gmdm.chainManager.ChainManagerEndInterface;
import midas.mw.mds.gmdm.exception.GmdmException;
import midas.mw.mds.internalinterfaces.MdsInternalInterface;
import midas.mw.mds.logging.MdsLogger;

public abstract class SyncManagerController
```

```
extends RequestHandler
implements SyncManagerInterface, ChainManagerEndInterface {

protected MdsInternalInterface mds;


protected Map configuration = new HashMap();
protected ChainManager chain;

protected boolean isResponseOnly = false;

public static ArrayList syncedWith = new ArrayList();
public static ArrayList respondedTo = new ArrayList();

public SyncManagerController(MdsInternalInterface mds) {
  super(mds);



  this.mds = mds;
  toComponent = AbstractMdsMessage.GMDM;

  chain = new ChainManager();
  setRequestId(String.valueOf(System.currentTimeMillis() +
      hashCode());

  configuration.put("node", this);

  GmdmSynchronize sync =null;
  try {
    sync = new GmdmSynchronize(mds);
  } catch (GmdmException e1) {
    MdsLogger.error("could not initiate the protocol:" + e1.
        getMessage(), this);
    MdsLogger.error(e1, this);

  }
  configuration.put("sync", sync);
  configuration.put("mds", mds);

  chain.setEnd(this);

}

/**
 * used to get this protocol runing as a thread, called only by
     start
 *
 * prefeard way to get the protocol running
 */
public void run()
{
  try {
    MdsLogger.info("Chain started", this);

    runChain();
  } catch (ChainException e) {
    MdsLogger.error(e, this);
  }
}

/**
```

```
   * runing the protocol best runed by invokeing of the start()
       method, to make a thread
   * @throws ChainException
   */
  public void runChain()
    throws ChainException
  {
    MdsLogger.debug("Running protocol " + getClass().getName() + "
        is response:" + isResponseOnly, this);

    this.configuration.put("protocollId", getRequestId());
    try {
      chain.runChain(this.configuration);
    } catch (ChainException e) {
      String nodeName = (String)configuration.get("toNode");
      if(e.getCause() instanceof MdsTimeoutException || ( e.
          getCause() != null && e.getCause().getCause() instanceof
           MdsTimeoutException) )
      {
        MdsLogger.info("Chain (with: "+ nodeName + ", " +
            getRequestId() + ") aborted by TimeOutException :" + e
            .getCause().getMessage() + ", " + nodeName + " reqestId
            :" + getRequestId() + " numberOfElementGained:" +
            configuration.get("addedMetadata"), this);

      }else if(e.getCause() instanceof ChainAbortedException || (e.
          getCause() != null && e.getCause().getCause() instanceof
          ChainAbortedException))
      {
        MdsLogger.info("Chain (with: "+ nodeName + ", " +
            getRequestId() + ") aborted by step with message:" + e.
            getMessage() + " >> "+ e.getCause().getMessage() + ", "
             + nodeName + " reqestId:" + getRequestId() + "
            numberOfElementGained:" + configuration.get("
            addedMetadata"), this);
      }
      else {
        MdsLogger.error("Chain (with: "+ nodeName + ", " +
            getRequestId() +") stoped unexpectedly, " + e.
            getMessage() + ", " + nodeName + " reqestId:" +
            getRequestId() + " numberOfElementGained:" +
            configuration.get("addedMetadata"), this);
        MdsLogger.error(e, this);
      }

    }
  }

  /**
   * called by any of the protocoll elements for getting the
       protocol to end
   * @param message
   * @param cause
   * @throws ChainException
   */
  public void abort(String message, Exception cause) throws
      ChainException {
    throw new ChainException("not in use", null);
  }

  /**
```

```
   * This is the standard implementation sync with every one in the
       neigbourhood list.
   * using the {@link #syncWith(String)} function for every node
   *
   * Overload if other action then defulte is required, for example
       {@link BroadCastSyncController#startSync()}
   *
   * @see SyncManagerInterface#startSync()
   */
  public void startSync()
    throws SyncManagerException
  {
    Vector nodes = mds.getCrt().getNeighbourhood();
    MdsLogger.info("started syncing with neighbors; " + nodes.
        toString() + " nodes", this);
    for(int i = 0; i < nodes.size(); i++)
      mds.getGmdm().getSyncManager().syncWith((String)nodes.get(i))
          ;
  }

  /**
   * called when the protocol is done
   * <p/>
   * must call {@link MdsDs#startLazySynch(Vector)} with the node
       that is synced
   * <p/>
   * can find the node wich was synced with via configuration.get("
       toNode")
   * @see #respondTo(GmdmMessage)
   * @see #syncWith(String)
   */
  public void end(Map configuration)
  {

    deregister();

    if(configuration == null)
    {

      return;
    }

    String syncedNode = (String)configuration.get("toNode");


    if(!isResponseOnly)
      MdsLogger.info("Sync prosess is finished, " + syncedNode + "
          reqestId:" + getRequestId() + " numberOfElementGained:" +
           (Integer)configuration.get("addedMetadata"), this);
    else
      MdsLogger.info("Sync response is finished, " + syncedNode + "
           reqestId:" + getRequestId() + " numberOfElementGained:"
          + (Integer)configuration.get("addedMetadata"), this);


    //if the toNode is set, then this is a protocoll wich do sync
        some thing
    // if not set this is typically a broadcast or one way sync, no
        results


    if(syncedNode != null)
    {
      //removing to only see the sm, protocol
      /*
      Vector node = new Vector();
      node.add(syncedNode);
      mds.getDs().startLazySynch(node);
      */
      //TODO this has to become more elegant

      try {
        if(configuration.get("addedMetadata") != null && ((Integer)
            configuration.get("addedMetadata")).intValue() > 0)
        {
          MdsLogger.info("starting a more sync to propagate new
              infor from " + syncedNode + " got " + ((Integer)
              configuration.get("addedMetadata")).intValue() + "
              elements", this);
          mds.getGmdm().newMetadataReceived(syncedNode);
        }
      } catch (GmdmException e) {
        MdsLogger.error(e, this);
      }

    }

  }

  /**
   * needs to set configuration["toNode"] to correct value
   *
   * @see super{@link #respondTo(GmdmMessage)}
   */
  public abstract void respondTo(GmdmMessage message) throws
      SyncManagerException ;

  /**
   * needs to set configuration["toNode"] to correct value
   *
   * @see super{@link #syncWith(String)}
   */
  public abstract void syncWith(String nodeId) throws
      SyncManagerException;

  /**
   * @see super{@link #triggerNewMetadata(String)}
   */
  public abstract void triggerNewMetadata(String fromNodeId) throws
       SyncManagerException ;

}
```

Listing B.4: The SyncManagerController is used to build synchronization protocols. As one can see the interface functions are abstract and need to be implemented by subclasses. Much of the class handles logging.

B.3 RollePlayer

```
package midas.apps.simulation;

```

```
import java.util.ArrayList;
import java.util.Map;
import java.util.TreeMap;
import java.util.Vector;

import midas.apps.simulation.actions.Action;
import midas.apps.simulation.actions.AllocateTableLocally;
import midas.apps.simulation.actions.MakeNewTableAction;
import midas.apps.simulation.actions.RemoteAllocate;
import midas.apps.simulation.actions.WhatDoGmdmKnowAction;
import midas.apps.simulation.actions.WhatsInsideTable;
import midas.mw.MwImpl;
import midas.mw.interfaces.Mw;
import midas.mw.mds.MdsCrtAdapter;
import midas.mw.mds.MdsCrtAdapterInterface;
import midas.mw.mds.MdsCrtCallBackAdapter;
import midas.mw.mds.MidasTextMessage;
import midas.mw.mds.XmlNode;
import midas.mw.mds.XmlParser;
import midas.mw.mds.internalinterfaces.MidasMessage;
import midas.mw.mds.logging.MdsLogger;
import midas.mw.mds.logging.StatusFile;
import midas.mw.mds.logging.StatusFileException;

public class RollePlayer implements MdsCrtCallBackAdapter {

  public long startTime;


  private ArrayList preformedActions = new ArrayList();
  Map actionMap = new TreeMap();

  Mw middleware;
  String nodeName;


  static Action[] actions = {
    new MakeNewTableAction(),
    new WhatDoGmdmKnowAction(),
    new AllocateTableLocally(),
    new MakeNewTableAction(),
    new RemoteAllocate(),
    new WhatsInsideTable()
  };
  /*  new InsertAction(),
    new InsertAnotherAction(),
    new UpdateAction()
  */


  /**
   * rolleplayer is a class which plays out some actions on a
       message initated bases.
   *
   * That is when a message send from the iemul is received at the
       rolleplayer the rolleplayer will check its
   * configuration and see for any actions to execute at this
       marker.
   *
   * @param nodeName name of the node
   * @param mapper marker to actions <String: marker name, Action:
       marker action>
```

```
     *
     * @see Action
     */
    public RollePlayer(String nodeName, Map mapper) {
      MdsLogger.setName(nodeName);
      MdsLogger.info("starting a RollePlayer on " + nodeName, this);
      actionMap = mapper;

      StatusFile.singleton = new StatusFile(nodeName + ".status");
      try {
        StatusFile.singleton.setStatusPoint("actionsPreformed",
            preformedActions, false);
      } catch (StatusFileException e) {
        MdsLogger.error(e, this);
      }
      /*
      Iterator keyActionMap = actionMap.keySet().iterator();
      while(keyActionMap.hasNext())
      {
        String marker = (String)keyActionMap.next();
        System.out.println("- '" + marker + "'");
        System.out.println(" -> '" + actionMap.get(marker).getClass()
            .getName());
      }
      */
      long milis = System.currentTimeMillis();
      MwImpl mw = new MwImpl(nodeName);
      mw.mwStart(mw);
      milis = System.currentTimeMillis() - milis;
      MdsLogger.error("Middleware ready in " + milis + "mili secs",
          this);


      middleware = mw;
      /*
       * for receiveing the messages form the sim
       */
      MdsCrtAdapterInterface crt = new MdsCrtAdapter(nodeName, "2006"
          , null);
      crt.start(mw);
      crt.registerCallBack(this);

      this.nodeName = nodeName;



    }


    public static void main(String[] args)
    {

      Map markerToActionMapper = new TreeMap();

      String nodeName = args[0];
      MdsLogger.setName(nodeName);
      XmlNode topNode = XmlParser.parse("senarioAction.xml").item(0);
      Vector markers = topNode.getChildren();
      for(int i = 0; i< markers.size(); i++)
      {
        XmlNode marker = (XmlNode)markers.get(i);
        Vector affectedNodes = marker.getChildren();
      for(int x = 0; x< affectedNodes.size(); x++)
      {
        XmlNode node = (XmlNode) affectedNodes.get(x);
        if(node.getAttribute("id").equals(nodeName) || node.
            getAttribute("id").startsWith("*"))
        {
          //action for this node or the all node char *
          markerToActionMapper.put(marker.getAttribute("id"),
              classFromText(node.getAttribute("action")));
          MdsLogger.debug("adding "+ marker.getAttribute("id") + "
              with object " + classFromText(node.getAttribute("
              action")), Thread.currentThread());
        }
      }
    }

    new RollePlayer(nodeName, markerToActionMapper);

  }


  private static Action classFromText(String attribute) {
    for(int i = 0; i< actions.length; i++)
    {
      if(attribute.equals(actions[i].getIdentifyer()))
        return actions[i];
    }
    return null;
  }


  /**
   * receiveing the messsages form the nett, or simulator and
       executing the correct
   * acrtion.
   */
  public void onReceivedMessage(MidasMessage message) {
    if(message instanceof MidasTextMessage)
    {
      MidasTextMessage msg = (MidasTextMessage) message;
      String marker = new String(msg.getText().toCharArray());

      //striping off any \0 at the end while this makes .equals go
          nuts
      for(int i = 0 ; i < marker.toCharArray().length; i++)
      {
        if(marker.charAt(i) == 0)
        {
          marker = marker.substring(0, i);
          i = marker.toCharArray().length;
        }
      }


      Action currentAction = (Action) actionMap.get(marker);

      try {
        currentAction.run(middleware);
        preformedActions.add(currentAction);//status file loging
        MdsLogger.info("Action found for node: " + nodeName + "
            marker: '" + marker + "'" , this);
      }catch(NullPointerException e){
        //MdsLogger.error(e, this);
        MdsLogger.info("*No* action found for node: " + nodeName +
            " marker: '" + marker + "'", this);

      }
      catch (Exception e) {
        MdsLogger.info("Error with node: " + nodeName + " marker: "
            + marker + " problem: " + e.getMessage(), this);
        MdsLogger.error(e, this);
      }
    }else
      MdsLogger.warn("dropping packet " + message.getClass().
          getName(), this);
  }

}
```

Listing B.5: The RollePlayer is the application doing the behaviour in our test scenarios. As one can see from the *onReceivedMessage()* any incoming message will result in running of an action, if the script says so.

B.4 Protocols

In this section all protocol source code is listed. First the protocol controller, followed by the steps.

B.4.1 SimpleSyncprotocol

```
package midas.mw.mds.gmdm.SyncManager;

import java.util.HashMap;
import java.util.Vector;

import midas.mw.mds.AbstractMdsMessage;
import midas.mw.mds.MdsCrtCallBackAdapter;
import midas.mw.mds.gmdm.GmdmMessage;
import midas.mw.mds.gmdm.SyncManager.steps.ReceiveAndHandleView;
import midas.mw.mds.gmdm.SyncManager.steps.ReceiveCompliment;
import midas.mw.mds.gmdm.SyncManager.steps.ReceiveNeeded;
import midas.mw.mds.gmdm.SyncManager.steps.SendView;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.gmdm.chainManager.ChainManagerEndInterface;
import midas.mw.mds.internalinterfaces.MdsInternalInterface;
import midas.mw.mds.logging.MdsLogger;


public class SimpleSyncProtocolController
  extends SyncManagerController
  implements MdsCrtCallBackAdapter, ChainManagerEndInterface,
      SyncManagerInterface {

  static int protocollCounter = 0;



  /**
```

```
     * Entry point for a sync, to comply with the {@link
         SyncManagerInterface}
     *
     * the chosen action must explicit be called {@link #respondTo(
         GmdmMessage)} or {@link #syncWith(String)}
     *
     * the
     *
     * @param mds
     */
    public SimpleSyncProtocolController(MdsInternalInterface mds)
    {
      super(mds);
      this.mds = mds;

      super.toComponent = AbstractMdsMessage.GMDM;

      MdsLogger.debug("made sync controller", this);
    }



    /**
     * @see SyncManagerInterface#respondTo(GmdmMessage);
     */
    public void respondTo(GmdmMessage msg) throws
        SyncManagerException {
      respondedTo.add(msg.getFromNode());

      isResponseOnly = true;
      setRequestId((String)msg.getRequestId());
      configuration.put("toNode", msg.getFromNode() );

      try {
        chain.add(new ReceiveAndHandleView());
        chain.add(new ReceiveNeeded());
      } catch (ChainException e) {
        MdsLogger.error(e, this);
        throw new SyncManagerException("Could not construct chain
            manager" , e);
      }


      //TODO this cant be right configuration = new TreeMap();



      MdsLogger.info("started a response on sync request form " + msg
          .getFromNode(), this);

      start();
    }

    /**
     * @see SyncManagerInterface#syncWith(String)
     */
    public void syncWith(String toNode) throws SyncManagerException {

      syncedWith.add(toNode);

      this.toComponent = AbstractMdsMessage.GMDM;

    try {
      chain.add(new SendView());
      chain.add(new ReceiveCompliment());
    } catch (ChainException e) {
      throw new SyncManagerException("Could not construct chain
          manager" , e);
    }


    //TODO this cant be right configuration = new TreeMap();
    configuration.put("toNode", toNode);


    MdsLogger.info("started a sync prosses with "+ toNode, this);

    start();

  }

  /**
   * @see SyncManagerInterface#triggerNewMetadata(String)
   */
  public void triggerNewMetadata(String fromNodeId)
    throws SyncManagerException {
    HashMap onlyonceMap = new HashMap();

    Vector nabours = mds.getCrt().getNeighbourhood();
    MdsLogger.debug("triggering on new metadata received from: " +
        fromNodeId + " nabours "+ nabours.toString() + " " +
        nabours.size(), this);
    for(int i = 0; i< nabours.size(); i++)
    {
      String nodeId = (String)nabours.get(i);
      if(!onlyonceMap.containsKey(nodeId))
      {
        onlyonceMap.put(nodeId, nodeId);
        if(!nodeId.equals(fromNodeId))
        {
          MdsLogger.debug("syncing with " + nodeId + " after new
              metadata", this);
          mds.getGmdm().getSyncManager().syncWith(nodeId); //
              constructing a new manager
        }
      }else
      {
        MdsLogger.warn("dual sync with " + nodeId + " on one
            triggerNewMetadata", this);
      }
    }

  }



}
```

Listing B.6: SimpleSyncProtocolController

```
package midas.mw.mds.gmdm.SyncManager.steps;
```

```

import java.util.Iterator;
import java.util.Map;
import java.util.Vector;

import midas.mw.mds.RequestHandler;
import midas.mw.mds.gmdm.SyncManager.messages.
    GmdmMetadataExchangeMessage;
import midas.mw.mds.gmdm.SyncManager.messages.
    GmdmSyncOverviewMesasge;
import midas.mw.mds.gmdm.chainManager.ChainElement;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.gmdm.exception.GmdmException;
import midas.mw.mds.internalinterfaces.MdsInternalInterface;
import midas.mw.mds.logging.MdsLogger;

/**
 * sending a total overview of the localy known instances to
     another node.
 *
 * <br/> much like the {@link BroadCastOverview} but this sends
     only to one other node
 * @author aslak
 *
 */
public class SendView extends ChainElement {

  GmdmMetadataExchangeMessage exchange = null;

  public void run(Map configuration) throws ChainException {
    RequestHandler handler = (RequestHandler)configuration.get("
        node");

    MdsInternalInterface mds = (MdsInternalInterface)configuration.
        get("mds");

    GmdmSyncOverviewMesasge message  = new GmdmSyncOverviewMesasge
        ((String)configuration.get("toNode"), mds);

    Vector knownElements = new Vector();
    Iterator it;
    try {
      it = mds.getGmdm().getSyncronizer().getOverview();
    } catch (GmdmException e) {
      throw new ChainException("could not get overview" , e);
    }
    while(it.hasNext())
    {
      String element = (String)it.next();
      MdsLogger.debug("element in overview: " + element, this);

      knownElements.add(element);
    }

    try {
      message.addAllKnownElement(knownElements);
    } catch (GmdmException e) {
      throw new ChainException("Could not add all known elemnts" ,
          e);
    }

    configuration.put("overview", message);
    MdsLogger.info("sending SyncOverview message to:" + message.
        getToNode(), this);
    handler.send(message);
  }



}
```

Listing B.7: SendView

```
package midas.mw.mds.gmdm.SyncManager.steps;

import java.util.Map;
import java.util.Vector;

import midas.mw.mds.MdsClientException;
import midas.mw.mds.MdsTimeoutException;
import midas.mw.mds.RequestHandler;
import midas.mw.mds.gmdm.GmdmSynchronize;
import midas.mw.mds.gmdm.SyncManager.messages.
    GmdmMetadataExchangeMessage;
import midas.mw.mds.gmdm.SyncManager.messages.
    GmdmSyncOverviewMesasge;
import midas.mw.mds.gmdm.chainManager.ChainAbortedException;
import midas.mw.mds.gmdm.chainManager.ChainElement;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.gmdm.exception.GmdmException;
import midas.mw.mds.internalinterfaces.MdsInternalInterface;
import midas.mw.mds.logging.MdsLogger;

/**
 * this step is used to respond to a overview message.
 * <br/>It requires that a message if type {@link
    GmdmSyncOverviewMesasge}  is received. The step will respond
    with
 * a {@link GmdmMetadataExchangeMessage} with the elements this
    node can provide
 * @author aslak
 *
 */
public class ReceiveAndHandleView extends ChainElement {



  public void run(Map configuration) throws ChainException {
    RequestHandler client = (RequestHandler)configuration.get("node
        ");

    GmdmSyncOverviewMesasge overview = null;
    try {
      //wait for the node to receve a message
      MdsLogger.debug("waiting for a GmdmSyncOverviewMessage", this
          );
      overview = (GmdmSyncOverviewMesasge)client.receive();

    } catch (MdsTimeoutException e) {
      MdsLogger.debug("time out on wait for overview", this);
      MdsLogger.info("excpected a overview did not receive", this);
      throw new ChainException("Timeout for receve", e);
    }catch (MdsClientException e)
    {
        MdsLogger.error("Having problems with the Request handler
            some thing whent wrong", this);
        throw new ChainException("Problems with the client and/or
            request handler",e);
      }

      GmdmSynchronize sync = (GmdmSynchronize)configuration.get("sync
          ");
      MdsInternalInterface mds = (MdsInternalInterface)configuration.
          get("mds");

      Vector data = null;
      try {
        data = sync.getCompliment(overview.getAllKnownElements());
      } catch (GmdmException e) {
        throw new ChainException("problems getting compliment", e);
      }

      MdsLogger.info("sending a MetadatExchange message in response
          to OverviewMessage", this);

      GmdmMetadataExchangeMessage message = new
          GmdmMetadataExchangeMessage(overview.getFromNode(), mds);
      message.addNeededElements(sync.getNeededElements(overview.
          getAllKnownElements()));
      MdsLogger.info("adding a request for needed elements count:"+
          message.getNeededElements().size(), this);
      try {
        message.addMetadataElements(data);
      } catch (GmdmException e1) {
        throw new ChainException("could not add the compliment
            elements", e1);
      }

      client.send(message);
      configuration.put("sendtMessage" , message);

      if(message.getNeededElements().isEmpty())
      {
        MdsLogger.info("Stoping the syncprotocol while there is no
            need for respnse to this message", this);
        throw new ChainAbortedException("No needed elements from " +
            message.getToNode() + ", no need for waiting for message"
            , null);
      }

    }

}
```

Listing B.8: ReceiveAndHandleView

```
package midas.mw.mds.gmdm.SyncManager.steps;

import java.util.Map;
import java.util.Vector;

import midas.mw.mds.MdsClientException;
import midas.mw.mds.MdsTimeoutException;
import midas.mw.mds.RequestHandler;
import midas.mw.mds.gmdm.GmdmImplementation;
import midas.mw.mds.gmdm.GmdmMetadata;
import midas.mw.mds.gmdm.GmdmSynchronize;
import midas.mw.mds.gmdm.SyncManager.SyncManagerController;
import midas.mw.mds.gmdm.SyncManager.SyncManagerException;
import midas.mw.mds.gmdm.SyncManager.messages.
    GmdmMetadataExchangeMessage;
import midas.mw.mds.gmdm.chainManager.ChainElement;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.gmdm.exception.GmdmException;
import midas.mw.mds.internalinterfaces.MdsInternalInterface;
import midas.mw.mds.internalinterfaces.MidasMessage;
import midas.mw.mds.logging.MdsLogger;

/**
 * This step will handel any {@link GmdmMetadataExchangeMessage}
     and will put the new elements
 * in the database in the correct way to inshure that all triggers
     is used properly.
 *
 * <br/> the step will also look for needed elements and send the
     elements needed to the needee in a {@link
     GmdmMetadataExchangeMessage}
 *
 *
 * @see GmdmImplementation#newMetadataReceived(String)
 * @author aslak
 *
 */
public class ReceiveCompliment extends ChainElement {


  public void run(Map configuration) throws ChainException {
    RequestHandler node = (RequestHandler) configuration.get("node"
        );

    GmdmMetadataExchangeMessage metaData =null;
    MdsInternalInterface mds = (MdsInternalInterface)configuration.
        get("mds");
      try {
        MdsLogger.debug("waiting for a MetaDataExchange message",
            this);
        MidasMessage tmpMessage= (MidasMessage)node.receive();

        //if this is a response to a loop back message, it is
            stoped here.
        if(tmpMessage.getFromNode().equals(mds.getNodeID()))
          throw new ChainException("Recevied message loopback,
              abort this handler", null);

        metaData = (GmdmMetadataExchangeMessage)tmpMessage;

      } catch (MdsTimeoutException e1) {
        throw new ChainException("Could not receve message before
            timeout" , e1);
      } catch (MdsClientException e1) {
        throw new ChainException("Could not receve message from
            node", e1);
      }



    GmdmSynchronize sync = (GmdmSynchronize) configuration.get("
        sync");
```

```
try {
  if(!metaData.getAllElements().isEmpty())
  {
    int numberOfElements = sync.instertSyncronation(metaData.
        getAllElements());
    if(numberOfElements > 0)
    {
      configuration.put("addedMetadata", new Integer(
          numberOfElements));

      //TODO remove for extra speed
      for(int i = 0; i < metaData.getAllElements().size(); i++)
        MdsLogger.debug("Receive metadata of table: " + ((
            GmdmMetadata)metaData.getAllElements().get(i)).
            getName(), this);

      //syncing this metadata out
      mds.getGmdm().newMetadataReceived(metaData.getFromNode())
          ;
    }else
      MdsLogger.debug("no elements is returned, nothing to get
          from other node:" + metaData.getFromNode(), this);
  }else
    MdsLogger.debug("no elements is returned, nothing to get
        from other node:" + metaData.getFromNode(), this);
} catch (GmdmException e) {
  throw new ChainException("Problem inserting meta data
      compliment", e);
}


// finding out whether we can provide some thing to the other
    side or not.
Vector needed = metaData.getNeededElements();
if(needed != null && !needed.isEmpty())
{
  MdsLogger.debug("Request for needed elements is found adding
      the requested elements", this);

  GmdmMetadataExchangeMessage msg = new
      GmdmMetadataExchangeMessage(metaData.getFromNode(), mds);
  try {
    Vector tmpVector = sync.getElementsByIdentifyer(needed);
    if(!tmpVector.isEmpty())
    {
      msg.addMetadataElements(tmpVector);
      MdsLogger.debug("sending a MetadataExchange message with
          response to needed elements count:" + msg.
          getAllElements().size(), this);
      node.send(msg);
    }else
      MdsLogger.warn("can not contribute with any element",
          this);

  } catch (GmdmException e) {
    MdsLogger.error(e, this);
  }



}else
```

```
      MdsLogger.info("No elements was requested through needed
          elements no mesage is sendt", this);

    //TODO: log the sync, how it whent and how many elemnts it
        synced
  }

}
```

Listing B.9: ReceiveCompliment

```
package midas.mw.mds.gmdm.SyncManager.steps;

import java.util.Map;

import midas.mw.mds.MdsImpl;
import midas.mw.mds.RequestHandler;
import midas.mw.mds.gmdm.GmdmSynchronize;
import midas.mw.mds.gmdm.SyncManager.SyncManagerException;
import midas.mw.mds.gmdm.SyncManager.messages.
    GmdmMetadataExchangeMessage;
import midas.mw.mds.gmdm.chainManager.ChainAbortedException;
import midas.mw.mds.gmdm.chainManager.ChainElement;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.gmdm.exception.GmdmException;
import midas.mw.mds.logging.MdsLogger;

/**
 * step receiveing all needed elements. will not make a nother
     message. Does the jobb of inserting all elements to
 * the database
 * @author aslak
 *
 */
public class ReceiveNeeded extends ChainElement {

  public void run(Map configuration)
    throws ChainException
  {
    GmdmMetadataExchangeMessage msg = (GmdmMetadataExchangeMessage)
        configuration.get("sendtMessage");

    if(msg.getNeededElements().isEmpty())
    {
      MdsLogger.info("Stoping the syncprotocol while there is no
          need for response to this message", this);
      throw new ChainAbortedException("No needed elements from " +
          msg.getToNode() + ", no need for waiting for message",
          new NotMoreElementsNeededException());
    }

    RequestHandler node = (RequestHandler) configuration.get("node"
        );
    MdsImpl mds = (MdsImpl) configuration.get("mds");
    GmdmMetadataExchangeMessage metaData =null;
    try {
      MdsLogger.debug("[" +this.hashCode() + "] waiting for
          requested needed elements", this);
      metaData = (GmdmMetadataExchangeMessage)node.receive();
    } catch (Exception e) {
      MdsLogger.debug("No message was receved with the needed
          elements", this);
      throw new ChainException("Problems receveing metadata
          iterator from node", e);
    }
    MdsLogger.debug("[" +this.hashCode() + "] message receved
        requId:" + metaData.getRequestId(), this);
    GmdmSynchronize sync = (GmdmSynchronize) configuration.get("
        sync");
    try {
      if(!metaData.getAllElements().isEmpty())
      {
        int numberOfElements = sync.instertSyncronation(metaData.
            getAllElements());
        if(numberOfElements > 0)
        {
          configuration.put("addedMetadata", new Integer(
              numberOfElements));

          mds.getGmdm().newMetadataReceived(metaData.getFromNode())
              ;
          MdsLogger.debug("receving number of needed elements:" +
              metaData.getAllElements().size(), this);
        }else
          MdsLogger.debug("recived empty needed message from " +
              metaData.getFromNode(), this);
      }

    } catch (GmdmException e) {
      throw new ChainException("Problem inserting meta data
          compliment", e);
    }
  }

}
```

Listing B.10: ReceiveNeeded

B.4.2 BroadcastProtocol

```
/**
 *
 */
package midas.mw.mds.gmdm.SyncManager;

import midas.mw.crtMock.BroadcastResponder;
import midas.mw.mds.AbstractMdsMessage;
import midas.mw.mds.gmdm.GmdmMessage;
import midas.mw.mds.gmdm.SyncManager.messages.
    BroadCastOverviewMessage;
import midas.mw.mds.gmdm.SyncManager.messages.ExchageRequestMessage
    ;
import midas.mw.mds.gmdm.SyncManager.steps.BroadCastOverview;
import midas.mw.mds.gmdm.SyncManager.steps.HandelExchangeMessage;
import midas.mw.mds.gmdm.SyncManager.steps.
    ReceiveBroadcastAndHandleView;
import midas.mw.mds.gmdm.SyncManager.steps.ReceiveDeliveryMesasge;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.internalinterfaces.MdsInternalInterface;
import midas.mw.mds.logging.MdsLogger;

/**
 * Controller for broadcast syncronation
 *
 * this will use the advantage of broadcast in wifi networks
 *
 * But while we are previously only having one to one sync will now
     have a one to many relationship
 * between the parties. This means that we need to have a seperate
     handling rutine for the response
 * of the BroadcastSyncOverview. This means that a overview is
     sendt, and responded with a ExchangeRequest
 * form every one that hears the overview and(&&) needs the
     elements provided.
 *
 * The exchange request at the initiating part can be received in
     the hundreds or none. This means that
 * the initiating part has to be ready to handle many or none. This
     will again put some restraint on the
 * protocoll.
 *
 * <p>
 * sender
 * BroadcastSyncOverviewMessage
 * on the create recieve eg.
 * triggers. The initiation part ends here and there is nothing
     left to receive the exchage requests
 * </p>
 *
 * <p>
 * receiver
 * Exchange Request
 * The receiver of a BroadcastSyncOverview has to 1) make a new
     request id. 2) send a message back to the sender
 * The replie in the Exchage Request is divided in two first the
     needed elements from the sender to the resiver(the request)
 * second the elements that the receiver can inlight the send about
     (provided)
 * </p>
 *
 * <p>
 * sender
 * DeliveryMessage
 * The sender is now in a new Request wich will handel all the
     ExchangeRequests. This will be handeld by giving the
 * receiver the elements that are in the request. This will be done
     in a DeliveryMessage. The provided elements is
 * added to the knowledge of the sender. This step is repeated for
     every ExchageRequest.
 * </p>
 *
 * @author aslak
 *
 */
public class BroadCastSyncController extends SyncManagerController{

  public BroadCastSyncController(MdsInternalInterface mds) {
    super(mds);
    toComponent = AbstractMdsMessage.GMDM;

    MdsLogger.debug("starting a broadcast sync prtocoll", this);

  }

  /* (non-Javadoc)
   * @see midas.mw.mds.gmdm.SyncManager.SyncManagerInterface#
       respondTo(midas.mw.mds.gmdm.GmdmMessage)
   */
  public void respondTo(GmdmMessage message) throws
      SyncManagerException {
    isResponseOnly = true;


    if(message.getFromNode().equals(mds.getNodeID()))
    {
      MdsLogger.debug("sync request from own node, dropping message
          ", this);
      return;
    }

    if (message instanceof BroadCastOverviewMessage) {
      try {

        chain.add(new ReceiveBroadcastAndHandleView());
        chain.add(new ReceiveDeliveryMesasge());
      } catch (ChainException e) {
        throw new SyncManagerException("could not construnct a
            handler chain for a Broadcast message", e);
      }

    }else if (message instanceof ExchageRequestMessage)
    {
      try {
        chain.add(new HandelExchangeMessage());
        setRequestId(message.getRequestId());
        //onReceivedMessage(message);
      } catch (ChainException e) {
        throw new SyncManagerException("could not construnct
            handler for exchangeMesasge", e);
      }
    }else
    {
      MdsLogger.warn("This message should not start a protocoll,
          dropping " + message.getClass().getName() + " req:" +
          message.getRequestId(), this);
    }

    configuration.put("toNode", message.getFromNode());

    MdsLogger.debug("Starting the protocoll in response to " +
        message.getClass().getName() +" from " + message.
        getFromNode(), this);
    start();
  }

  /* (non-Javadoc)
   * @see midas.mw.mds.gmdm.SyncManager.SyncManagerInterface#
       syncWith(java.lang.String)
   */
  /**
   * using the {@link SimpleSyncProtocolController#syncWith(String)
       } function, while this will be the same
   */
  public void syncWith(String nodeId) throws SyncManagerException {

    syncWithEveryOne();
```

```
	}

	/* (non-Javadoc)
	 * @see midas.mw.mds.gmdm.SyncManager.SyncManagerInterface#
	     triggerNewMetadata(java.lang.String)
	 */
	public void triggerNewMetadata(String fromNodeId)
		throws SyncManagerException {
		syncWithEveryOne();
	}

	public void startSync()
		throws SyncManagerException
	{
		syncWithEveryOne();
	}

	private void syncWithEveryOne()
		throws SyncManagerException
	{
		syncedWith.add("everyOne");
		try {
			chain.add(new BroadCastOverview());
		} catch (ChainException e) {
			throw new SyncManagerException("Error in sync protocoll
			    execution", e);
		}

		chain.setEnd(this);
		MdsLogger.info("Starting the protocoll with every one", this);
		start();
	}


}
```

Listing B.11: BroadCastSyncController

```
package midas.mw.mds.gmdm.SyncManager.steps;

import java.util.Iterator;
import java.util.Map;

import midas.mw.mds.RequestHandler;
import midas.mw.mds.gmdm.GmdmSynchronize;
import midas.mw.mds.gmdm.SyncManager.messages.
    BroadCastOverviewMessage;
import midas.mw.mds.gmdm.chainManager.ChainElement;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.gmdm.exception.GmdmException;
import midas.mw.mds.internalinterfaces.MdsInternalInterface;
import midas.mw.mds.logging.MdsLogger;

/**
 * this step is taking advantage of the broadcast capabilitys in
     wifi networks. This will save bandwith while the
 * step is only sending one message to all the neighbours.
 *
 *
 * @author aslak
 *
 */
```

```
public class BroadCastOverview extends ChainElement {

  public void run(Map config) throws ChainException
  {
    RequestHandler handler = (RequestHandler) config.get("node");
    MdsInternalInterface mds = (MdsInternalInterface) config.get("
        mds");

    BroadCastOverviewMessage message = new BroadCastOverviewMessage
        (mds);
    GmdmSynchronize sync = (GmdmSynchronize) config.get("sync");

    try {
      Iterator elements = sync.getOverview();
      while(elements.hasNext())
      {
        message.addKnownElement((String) elements.next());
      }

    } catch (GmdmException e) {
      throw new ChainException("error in adding overview", e);
    }

    MdsLogger.debug("sending broadcast message", this);
    handler.send(message);
  }

}
```

Listing B.12: BroadCastOverview

```
package midas.mw.mds.gmdm.SyncManager.steps;

import java.util.Map;
import java.util.Vector;

import midas.mw.mds.MdsClientException;
import midas.mw.mds.MdsTimeoutException;
import midas.mw.mds.RequestHandler;
import midas.mw.mds.gmdm.GmdmSynchronize;
import midas.mw.mds.gmdm.SyncManager.SyncManagerController;
import midas.mw.mds.gmdm.SyncManager.messages.
    BroadCastOverviewMessage;
import midas.mw.mds.gmdm.SyncManager.messages.ExchageRequestMessage
    ;
import midas.mw.mds.gmdm.chainManager.ChainElement;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.gmdm.exception.GmdmException;
import midas.mw.mds.internalinterfaces.MdsInternalInterface;
import midas.mw.mds.logging.MdsLogger;

/**
 * This step is designed to handel the {@link
    BroadCastOverviewMessage} and then if there is something
    requested wait for the respondes.
 * The step is to responde to the message with a exchange message
    if <b> and only if</b> this node can
 * provied or need some elements. This will be sendt trough a {
    @link ExchageRequestMessage}
 *
 * @author aslak
 *
 */
public class ReceiveBroadcastAndHandleView extends ChainElement {


  public void run(Map configuration) throws ChainException {
    RequestHandler client = (RequestHandler)configuration.get("node
        ");

    BroadCastOverviewMessage overview = null;
    try {
      //wait for the node to receve a message
      MdsLogger.debug("waiting for a BroadcastOverviewMessage" ,
          this);
      overview = (BroadCastOverviewMessage)client.receive();

      MdsLogger.debug("BroadcastResponse new Seq: " + client.
          getRequestId() + " old:"+ overview.getRequestId(), this)
          ;

    } catch (MdsTimeoutException e) {
      MdsLogger.debug("time out on wait for overview", this);
      MdsLogger.info("excpected a overview did not receive", this);
      throw new ChainException("Timeout for receve", e);
    }catch (MdsClientException e)
    {
      MdsLogger.error("Having problems with the Request handler
          some thing whent wrong", this);
      throw new ChainException("Problems with the client and/or
          request handler",e);
    }

    GmdmSynchronize sync = (GmdmSynchronize)configuration.get("sync
        ");
    MdsInternalInterface mds = (MdsInternalInterface)configuration.
        get("mds");

    Vector data = null;
    try {
      data = sync.getCompliment(overview.getAllKnownElements());
    } catch (GmdmException e) {
      throw new ChainException("problems getting compliment", e);
    }


    ExchageRequestMessage message = new ExchageRequestMessage(
        overview.getFromNode(), mds);
    message.addRequest(sync.getNeededElements(overview.
        getAllKnownElements()));

    MdsLogger.info("Requesting needed elements "+ message.
        getRequest().toString(), this);
    try {
      message.addProvided(data);
    } catch (GmdmException e1) {
      throw new ChainException("could not add the provided elements
          ", e1);
    }


    if(message.getRequest().isEmpty())
    {
      //save the wifi net, and abort the protocoll here
      MdsLogger.info("No info needed from " + message.getFromNode()
          , this);
      throw new ChainException("Aborted due to non needed", new
          NotMoreElementsNeededException());
    }else
    {
      MdsLogger.warn("Responding to over view, by requesting; " +
          message.getRequest() + " providing; "+ message.
          getProvided(), this);
    }

    client.send(message);
    SyncManagerController.respondedTo.add(message.getFromNode());
    MdsLogger.info("sending a ExchangeRequest("+ message.
        getProvided().toString() + " wanted: "+ message.getRequest
        ().toString() + ") message in response to
        BroadcastOverviewMessage", this);
    configuration.put("requestCount", new Integer(message.
        getRequest().size()));

    /*
    //receiveing the bcast message before we terminate the requid
    try {
      client.receive();
    } catch (MdsTimeoutException e) {
      MdsLogger.warn("the loopback broadcast message could not be
          received in time out", this);
      MdsLogger.warn(e, this);
    } catch (MdsClientException e) {
      MdsLogger.warn("the loopback broadcast message could not be
          received", this);
      MdsLogger.warn(e, this);
    }
    */
  }

}
```

Listing B.13: ReceiveBroadcastAndHandleView

```
package midas.mw.mds.gmdm.SyncManager.steps;

import java.util.Map;

import midas.mw.mds.MdsClientException;
import midas.mw.mds.MdsTimeoutException;
import midas.mw.mds.RequestHandler;
import midas.mw.mds.gmdm.GmdmSynchronize;
import midas.mw.mds.gmdm.SyncManager.messages.DeliveryMessage;
import midas.mw.mds.gmdm.chainManager.ChainElement;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.gmdm.exception.GmdmException;
import midas.mw.mds.logging.MdsLogger;

public class ReceiveDeliveryMesasge extends ChainElement {

  public void run(Map config)
    throws ChainException
  {
    if(((Integer)config.get("requestCount")).intValue() > 0)
    {
      RequestHandler handler = (RequestHandler) config.get("node");
      DeliveryMessage message;
      try {
        message = (DeliveryMessage) handler.receive();
      } catch (MdsTimeoutException e) {
        throw new ChainException("could not get message before time
            out", e);
      } catch (MdsClientException e) {
        throw new ChainException("could not receive message", e);
      }
      MdsLogger.debug("got delivery " + message.getData().toString
          (), this);
      GmdmSynchronize sync = (GmdmSynchronize)config.get("sync");
      try {
        int numberOfElements = sync.instertSyncronation(message.
            getData());
        if(numberOfElements > 0 )
          config.put("addedMetadata", new Integer(numberOfElements)
              );

        MdsLogger.info("Recived needed metadata("+ message.getData
            ().toString() +") from "+ message.getFromNode(), this);
      } catch (GmdmException e) {
        throw new ChainException("could not insert the new elements
            ", e);
      }
    }else
    {
      MdsLogger.info("The sync prosess is not requesting any thing
          and is there for ended", this);
    }
  }
}
```

Listing B.14: ReceiveDeliveryMesasge

```
package midas.mw.mds.gmdm.SyncManager.steps;

import java.util.Map;

import midas.mw.mds.MdsClientException;
import midas.mw.mds.MdsTimeoutException;
import midas.mw.mds.RequestHandler;
import midas.mw.mds.gmdm.GmdmSynchronize;
import midas.mw.mds.gmdm.SyncManager.messages.DeliveryMessage;
import midas.mw.mds.gmdm.SyncManager.messages.ExchageRequestMessage
    ;
import midas.mw.mds.gmdm.chainManager.ChainElement;
import midas.mw.mds.gmdm.chainManager.ChainException;
import midas.mw.mds.gmdm.exception.GmdmException;
import midas.mw.mds.internalinterfaces.MdsInternalInterface;

public class HandelExchangeMessage extends ChainElement {

  public void run(Map configuration)
    throws ChainException
  {
    RequestHandler client = (RequestHandler)configuration.get("node
        ");
    GmdmSynchronize sync = (GmdmSynchronize)configuration.get("sync
        ");
    MdsInternalInterface mds = (MdsInternalInterface)configuration.
        get("mds");

    ExchageRequestMessage message;
    try {
      message = (ExchageRequestMessage) client.receive();
    } catch (MdsTimeoutException e) {
      throw new ChainException("could not get message before time
          out", e);
    } catch (MdsClientException e) {
      throw new ChainException("could not receive message", e);
    }

    try {
      int numberOfElements = sync.instertSyncronation(message.
          getProvided());
      if(numberOfElements > 0 )
          configuration.put("addedMetadata", new Integer(
              numberOfElements));

    } catch (GmdmException e) {
      throw new ChainException("could not insert new metadata", e);
    }



    if(message.getRequest().size() > 0)
    {
      DeliveryMessage delivery = new DeliveryMessage(message.
          getFromNode(), mds);
      try {
        delivery.addData(sync.getElementsByIdentifyer(message.
            getRequest()));
      } catch (GmdmException e) {
        throw new ChainException("could not get the requested
            elements",e);
      }

      client.send(delivery);
    }

  }
}
```

Listing B.15: HandelExchangeMessage

B.4.3 SemanticSyncProtocol

```
/**
 *
 */
package midas.mw.mds.gmdm.SyncManager;

import java.util.Vector;

import midas.mw.mds.MdsClientException;
import midas.mw.mds.MdsTimeoutException;
import midas.mw.mds.XmlNode;
import midas.mw.mds.XmlParser;
import midas.mw.mds.gmdm.GmdmMessage;
```

```java
import midas.mw.mds.gmdm.SyncManager.messages.
    BroadCastOverviewMessage;
import midas.mw.mds.gmdm.SyncManager.messages.ExchageRequestMessage
    ;
import midas.mw.mds.gmdm.SyncManager.messages.
    GmdmMetadataExchangeMessage;
import midas.mw.mds.gmdm.SyncManager.messages.
    GmdmSyncOverviewMesasge;
import midas.mw.mds.internalinterfaces.MdsInternalInterface;
import midas.mw.mds.internalinterfaces.MidasMessage;
import midas.mw.mds.logging.MdsLogger;

/**
 * This protocol is to use the advantage of the group which the
     user is part of.
 *
 * The protocol is to try to get higher speed of the meta data
     spred. So that all
 * the elements in shortest time possible achives 100% coverage.
     This will be node in the following way
 *
 *
 *     sender            receiver                        sender
 * SemanticOverviewMessage >>   @link {@link
     GmdmMetadataExchangeMessage} >> {@link
     GmdmMetadataExchangeMessage}
 * A message to all in same      a response with elements and needed
          the needed elements
 * group as sender
 *
 *
 * @author aslak
 *
 */
public class SemanticSyncController extends SyncManagerController{

  String groupName = "";
  Vector groupMembers = new Vector(0);

  public SemanticSyncController(MdsInternalInterface mds) {
    super(mds);

    initializeGroup();
  }



  /**
   * Using broadcast protocol to respond to
       BroadCastOverviewMessage and ExchageRequestMessage <br/>
   * and simplecontroller to GmdmSyncOverviewMesasge
   * @see midas.mw.mds.gmdm.SyncManager.SyncManagerInterface#
       respondTo(midas.mw.mds.gmdm.GmdmMessage)
   */
  public void respondTo(GmdmMessage message) throws
      SyncManagerException {

    MdsLogger.debug("responding to " + message.getClass().getName()
        + " req: " + message.getRequestId() + " from " + message.
        getFromNode(), this);
    SyncManagerController controller  = null;
    if(message instanceof BroadCastOverviewMessage)
    {
      //a response is needed for a broadcast message
      controller = new BroadCastSyncController(mds);
    }else if(message instanceof GmdmSyncOverviewMesasge)
    {
      //a response is needed for a simple sync
      controller = new SimpleSyncProtocolController(mds);
    }else if( message instanceof ExchageRequestMessage)
    {
      //a response to the previous broadcast message
      controller = new BroadCastSyncController(mds);
    }else
    {
      MdsLogger.warn("message not recognised " + message.getClass()
          .getName() + " req:" + message.getRequestId()   , this);
      return;
    }

    try {
      MidasMessage msg = receive();
      MdsLogger.debug("passed on message " + msg.getClass().getName
          () +
          " req:" + msg.getRequestId() +
          "to " + controller.getClass().getName()
          , this
          );
      controller.onReceivedMessage(msg);

    } catch (MdsTimeoutException e) {
      MdsLogger.debug("sould not be waiting this is only forward",
          this);
    } catch (MdsClientException e) {
      MdsLogger.debug(e, this);
    }

    controller.respondTo(message);
    end(null);
  }

  /**
   * starting sync with one node using SimpleSyncProtocolController
   * @see midas.mw.mds.gmdm.SyncManager.SyncManagerInterface#
       syncWith(java.lang.String)
   */
  public void syncWith(String nodeId) throws SyncManagerException {
    if(nodeId.equals(mds.getNodeID()))
    {
      MdsLogger.debug("refusing to sync with itself!", this);
      return;
    }

    MdsLogger.debug("syncing with " + nodeId  , this);
    SimpleSyncProtocolController simple = new
        SimpleSyncProtocolController(mds);
    simple.syncWith(nodeId);
  }

  /**
   * Starting the {@link #startSync()}
   * @see midas.mw.mds.gmdm.SyncManager.SyncManagerInterface#
       triggerNewMetadata(java.lang.String)
   */
  public void triggerNewMetadata(String fromNodeId)
      throws SyncManagerException {
    MdsLogger.debug("triggerd on metadata", this);

    startSync();

  }

  /**
   * syncing with all in this group. could be none.
   * using {@link #syncWith(String)}
   */
  public void syncWithGroup()
  {
    MdsLogger.debug("syncing with the neigbours one by one " +
        groupMembers, this);
    if(groupMembers.isEmpty())
    {
      MdsLogger.info("no one in your group", this);
      return;
    }
    for(int i = 0 ; i < groupMembers.size(); i++)
    {
      try {
        syncWith((String) groupMembers.get(i));
      } catch (SyncManagerException e) {
        MdsLogger.error(e, this);
      }
    }
  }

  /**
   * syncing with all the neigbours defined by all that hears the {
       @link BroadCastOverviewMessage} via {@link
       BroadCastSyncController}
   *
   */
  public void syncWithNeigbours()
  {
    MdsLogger.debug("syncing with all the neigbours", this);
    BroadCastSyncController bcast = new BroadCastSyncController(mds
        );
    try {
      bcast.startSync();
    } catch (SyncManagerException e) {
      MdsLogger.error(e, this);
    }
  }

  /**
   * reading the groups.xml and finding out wich are in the same
       group
   *
   */
  public void initializeGroup()
  {

    Vector groups;
    try {
      groups = ((XmlNode) XmlParser.parse("groups.xml").getChildren
          ().get(0)).getChildren();
    } catch (Exception e) {
      MdsLogger.debug("This node is in no group", this);
      groupMembers = new Vector();
      groupName ="single";
      return;
    }

    for(int i = 0; i < groups.size(); i++ )
    {
      Vector group = ((XmlNode)groups.get(i)).getChildren();

      Vector members =  new Vector();
      for(int x = 0; x < group.size(); x++)
      {
        String name = ((XmlNode)group.get(x)).getAttribute("nodeId"
            );
        members.add(name);
      }
      if(members.contains(mds.getNodeID()))
      {
        groupMembers = members;
        groupName = ((XmlNode)groups.get(i)).getAttribute("id");
        break;
      }
    }

    MdsLogger.debug("your group " + groupName + ", " + groupMembers
        , this);
  }

  /**
   * starting sync, by doing {@link #syncWithGroup()} and then {
       @link #syncWithNeigbours()}
   * @see midas.mw.mds.gmdm.SyncManager.SyncManagerController#
       startSync()
   */
  public void startSync()
  {
    MdsLogger.debug("started full sync" ,this);
    syncWithGroup();
    syncWithNeigbours();
  }

}
```

Listing B.16: SemanticSyncController

Appendix C

UML model MIDAS

I will now display the componet design of MDS and GMDM.

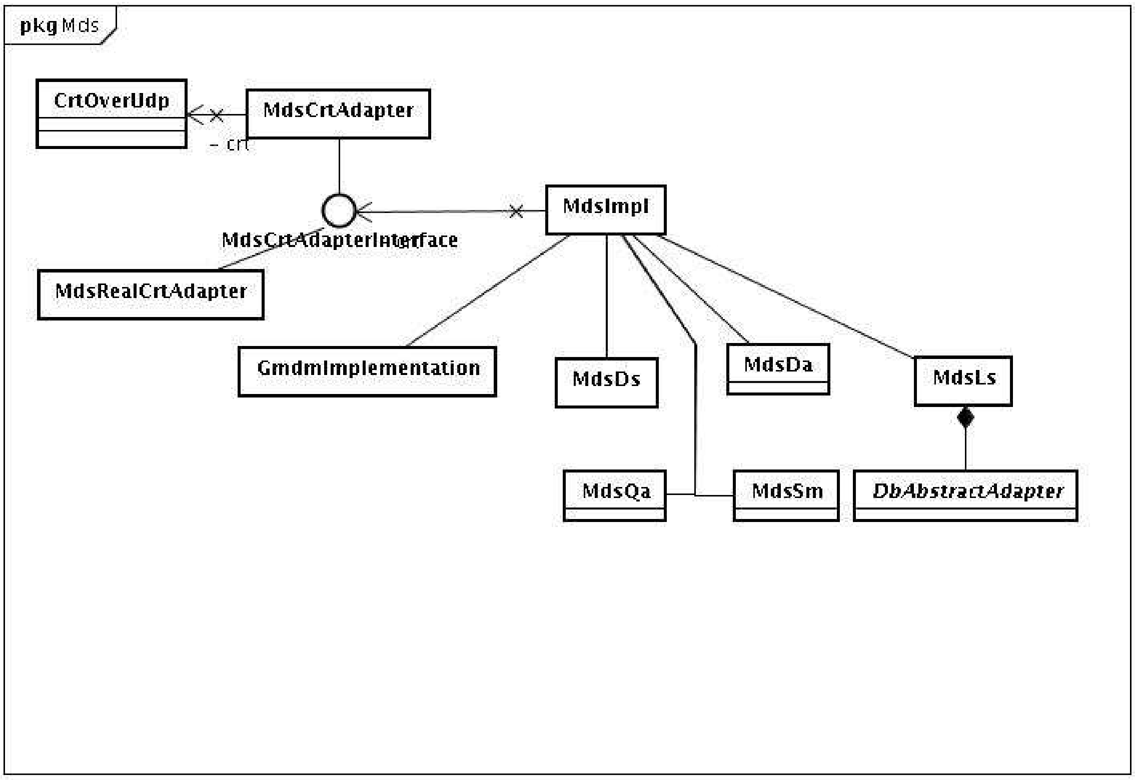


Figure C.1: Overview of MDS component in MIDAS

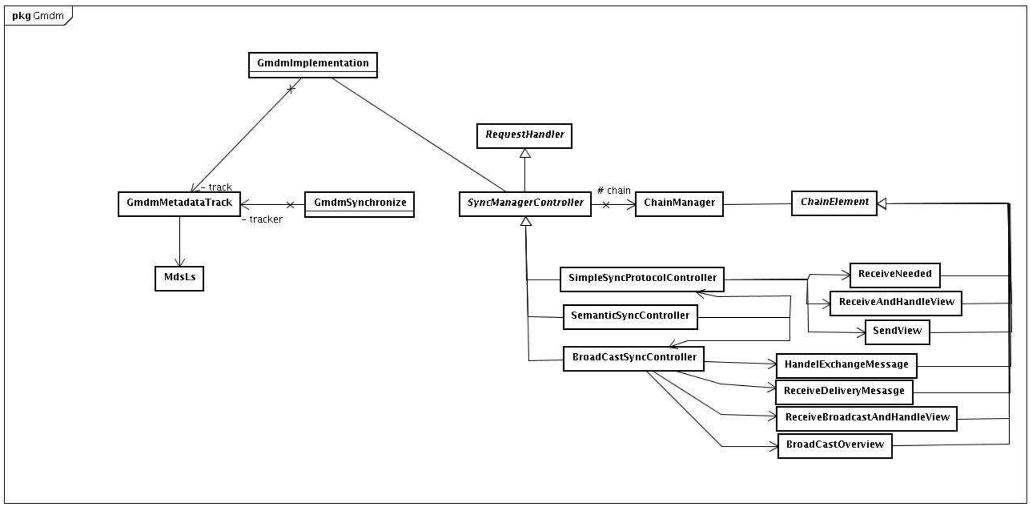


Figure C.2: Uml diagram of the GMDM component

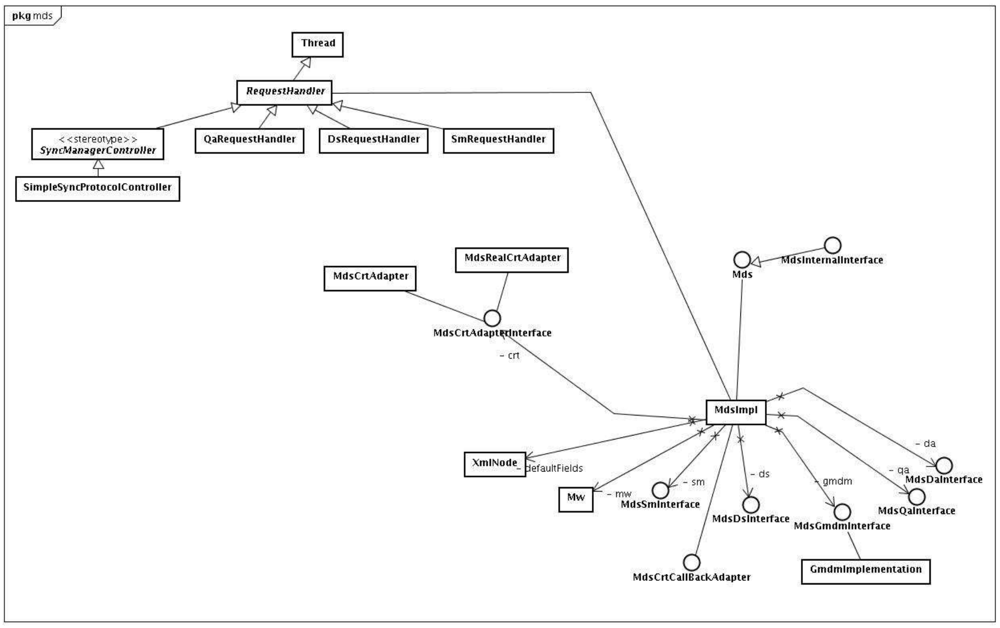


Figure C.3: Uml diagram showing the internal structure of Mds

Appendix D

UML model Infoware implementation

In this appendix the overall architecture of the DENS implementation is presented.

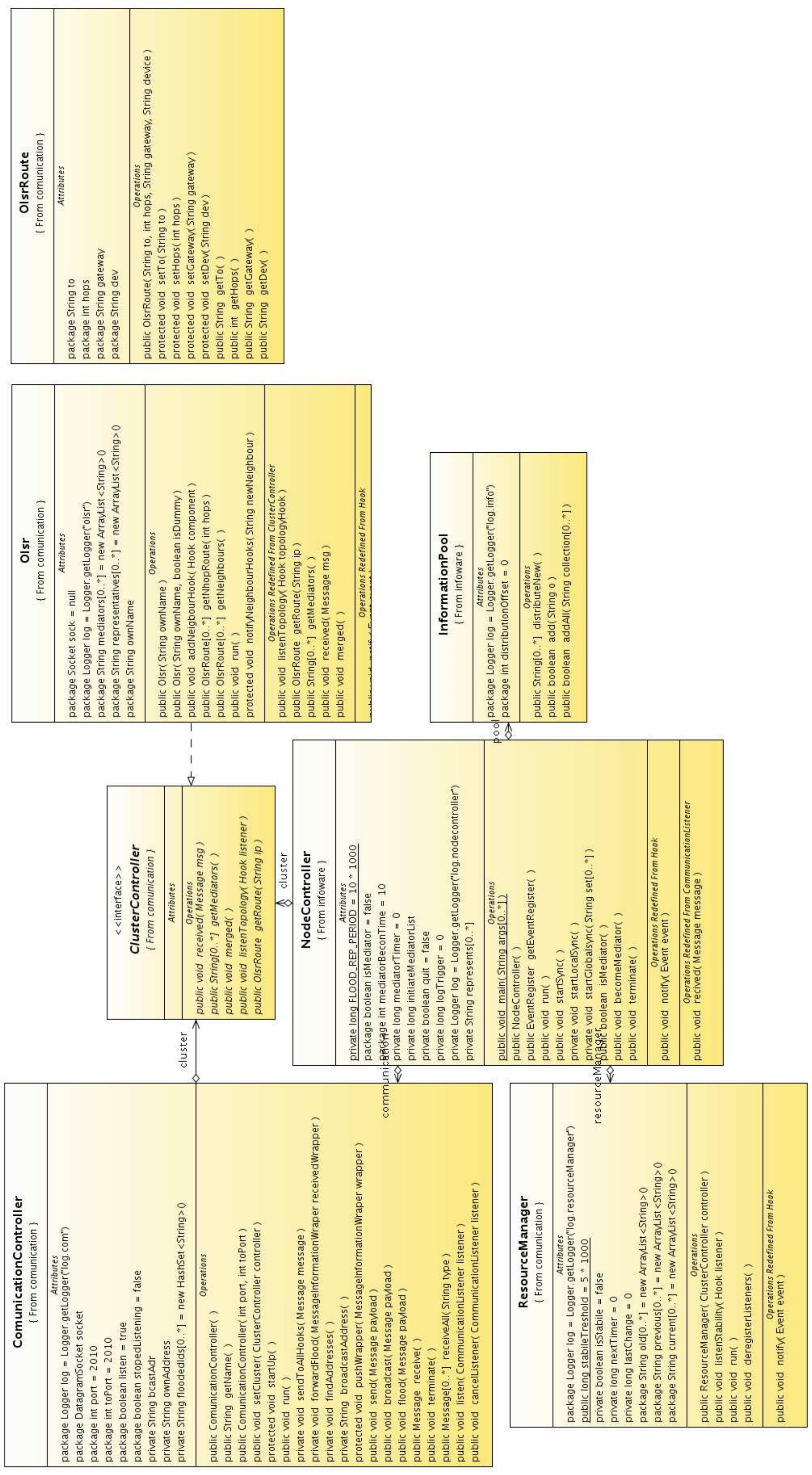

OlsrRoute
{ From comunication }
Attributes
package String to
package int hops
package String gateway
package String dev
Operations
public OlsrRoute(String to, int hops, String gateway, String device)
protected void setTo(String to)
protected void setHops(int hops)
protected void setGateway(String gateway)
protected void setDev(String dev)
public String getTo( )
public int getHops( )
public String getGateway( )
public String getDev( )
Olsr
{ From comunication }
Attributes
package Socket sock = null
package Logger log = Logger.getLogger("olsr")
package String mediators[0..*] = new ArrayList<String>()
package String representatives[0..*] = new ArrayList<String>()
package String ownName
Operations
public Olsr(String ownName)
public Olsr(String ownName, boolean isDummy)
public void addNeigbourHook(Hook component)
public OlsrRoute[0..*] getNhopRoute(int hops)
public OlsrRoute[0..*] getNeighbours( )
public void run( )
protected void notifyNeighbourHooks(String newNeighbour)
Operations Redefined From ClusterController
public void listenTopology(Hook topologyHook)
public OlsrRoute getRoute(String ip)
public String[0..*] getMediators( )
public void received(Message msg)
public void merged( )
Operations Redefined From Hook
InformationPool
{ From infoware }
Attributes
package Logger log = Logger.getLogger("log.info")
package int distributionOffset = 0
Operations
public String[0..*] distributeNew( )
public boolean add(String o)
public boolean addAll(String collection[0..*])
pool
<<interface>>
ClusterController
{ From comunication }
Attributes
Operations
public void received(Message msg)
public String[0..*] getMediators( )
public void merged( )
public void listenTopology(Hook listener)
public OlsrRoute getRoute(String ip)
cluster
cluster
NodeController
{ From infoware }
Attributes
private long FLOOD_REP_PERIOD = 10 * 1000
package boolean isMediator = false
package int mediatorBeaconTime = 10
private long mediatorTimer = 0
private long initiateMediatorList
private boolean quit = false
private long logTrigger = 0
private Logger log = Logger.getLogger("log.nodecontroller")
private String represents[0..*]
Operations
public void main(String args[0..*])
public NodeController( )
public EventRegister getEventRegister( )
public void run( )
public void startSync( )
private void startLocalSync( )
private void startGlobalsync(String set[0..*])
public boolean isMediator( )
public void becomeMediator( )
public void terminate( )
Operations Redefined From Hook
public void notify(Event event)
Operations Redefined From CommunicationListener
public void recived(Message message)
communication
resourceManager
ComunicationController
{ From comunication }
Attributes
package Logger log = Logger.getLogger("log.com")
package DatagramSocket socket
package int port = 2010
package int toPort = 2010
package boolean listen = true
package boolean stopedListening = false
private String bcastAdr
private String ownAddress
private String floodedIds[0..*] = new HashSet<String>()
Operations
public ComunicationController( )
public String getName( )
public ComunicationController(int port, int toPort)
public void setCluster(ClusterController controller)
protected void startUp( )
public void run( )
private void sendToAllHooks(Message message)
private void forwardFlood(MessageInformationWraper receivedWrapper)
private void findAddresses( )
private String broadcastAddress( )
protected void pushWrapper(MessageInformationWraper wrapper)
public void send(Message payload)
public void broadcast(Message payload)
public void flood(Message payload)
public Message receive( )
public void terminate( )
public Message[0..*] receiveAll(String type)
public void listen(CommunicationListener listener)
public void cancelListener(CommunicationListener listener)
ResourceManager
{ From comunication }
Attributes
package Logger log = Logger.getLogger("log.resourceManager")
public long stabileTreshold = 5 * 1000
private boolean isStabile = false
private long nextTimer = 0
private long lastChange = 0
package String old[0..*] = new ArrayList<String>()
package String previous[0..*] = new ArrayList<String>()
package String current[0..*] = new ArrayList<String>()
Operations
public ResourceManager(ClusterController controller)
public void listenStability(Hook listener)
public void run( )
public void deregisterListeners( )
Operations Redefined From Hook
public void notify(Event event)

Figure D.1: The UML model showing the overview of DENS architecture.

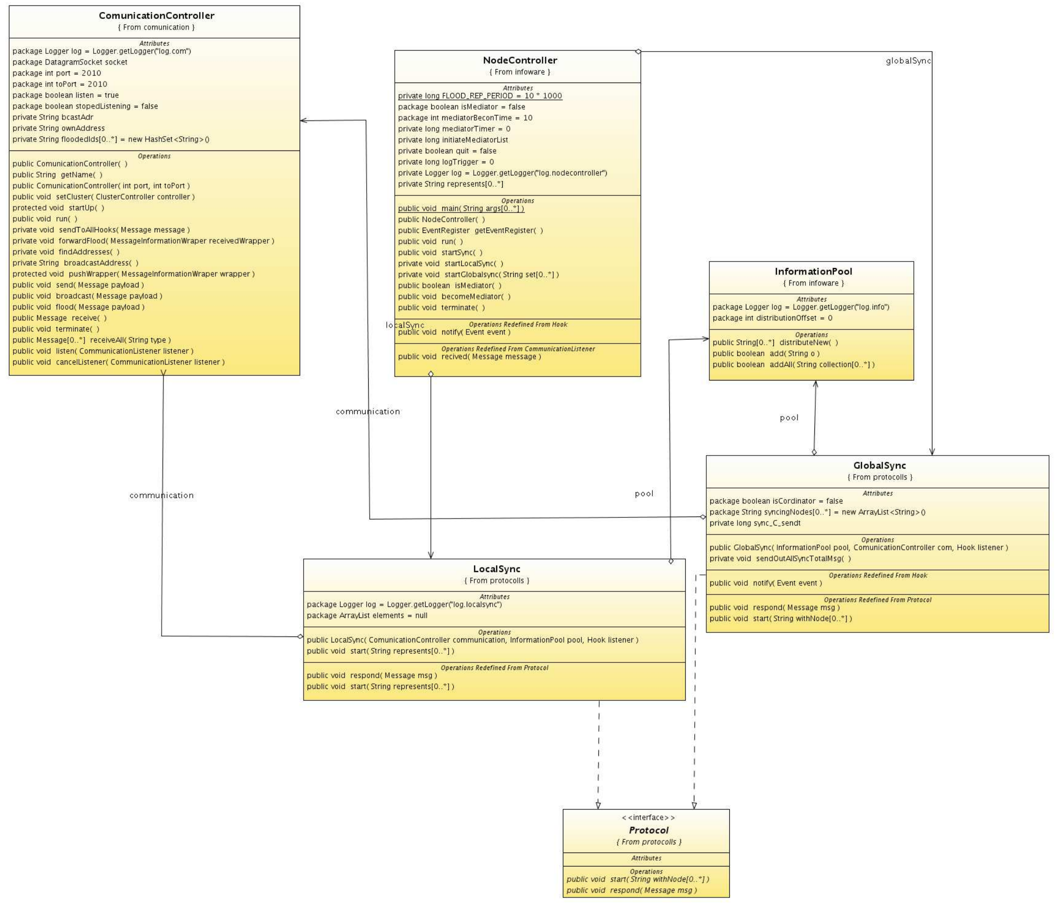

ComunicationController
{ From comunication }
Attributes
package Logger log = Logger.getLogger("log.com")
package DatagramSocket socket
package int port = 2010
package int toPort = 2010
package boolean listen = true
package boolean stopedListening = false
private String bcastAdr
private String ownAddress
private String floodedIds[0..*] = new HashSet<String>()
Operations
public ComunicationController( )
public String getName( )
public ComunicationController(int port, int toPort)
public void setCluster(ClusterController controller)
protected void startUp( )
public void run( )
private void sendToAllHooks(Message message)
private void forwardFlood(MessageInformationWraper receivedWrapper)
private void findAddresses( )
private String broadcastAddress( )
protected void pushWrapper(MessageInformationWraper wrapper)
public void send(Message payload)
public void broadcast(Message payload)
public void flood(Message payload)
public Message receive( )
public void terminate( )
public Message[0..*] receiveAll(String type)
public void listen(CommunicationListener listener)
public void cancelListener(CommunicationListener listener)
NodeController
{ From infoware }
Attributes
private long FLOOD_REP_PERIOD = 10 * 1000
package boolean isMediator = false
package int mediatorBeconTime = 10
private long mediatorTimer = 0
private long initiateMediatorList
private boolean quit = false
private long logTrigger = 0
private Logger log = Logger.getLogger("log.nodecontroller")
private String represents[0..*]
Operations
public void main(String args[0..*])
public NodeController( )
public EventRegister getEventRegister( )
public void run( )
public void startSync( )
private void startLocalSync( )
private void startGlobalsync(String set[0..*])
public boolean isMediator( )
public void becomeMediator( )
public void terminate( )
Operations Redefined From Hook
public void notify(Event event)
Operations Redefined From CommunicationListener
public void recived(Message message)
globalSync
localSync
communication
InformationPool
{ From infoware }
Attributes
package Logger log = Logger.getLogger("log.info")
package int distributionOffset = 0
Operations
public String[0..*] distributeNew( )
public boolean add(String o)
public boolean addAll(String collection[0..*])
pool
communication
pool
GlobalSync
{ From protocolls }
Attributes
package boolean isCordinator = false
package String syncingNodes[0..*] = new ArrayList<String>()
private long sync_C_sendt
Operations
public GlobalSync(InformationPool pool, ComunicationController com, Hook listener)
private void sendOutAllSyncTotalMsg( )
Operations Redefined From Hook
public void notify(Event event)
Operations Redefined From Protocol
public void respond(Message msg)
public void start(String withNode[0..*])
LocalSync
{ From protocolls }
Attributes
package Logger log = Logger.getLogger("log.localsync")
package ArrayList elements = null
Operations
public LocalSync(ComunicationController communication, InformationPool pool, Hook listener)
public void start(String represents[0..*])
Operations Redefined From Protocol
public void respond(Message msg)
public void start(String represents[0..*])
<<interface>>
Protocol
{ From protocolls }
Attributes
Operations
public void start(String withNode[0..*])
public void respond(Message msg)

Figure D.2: The UML model showing the DENS protocols.

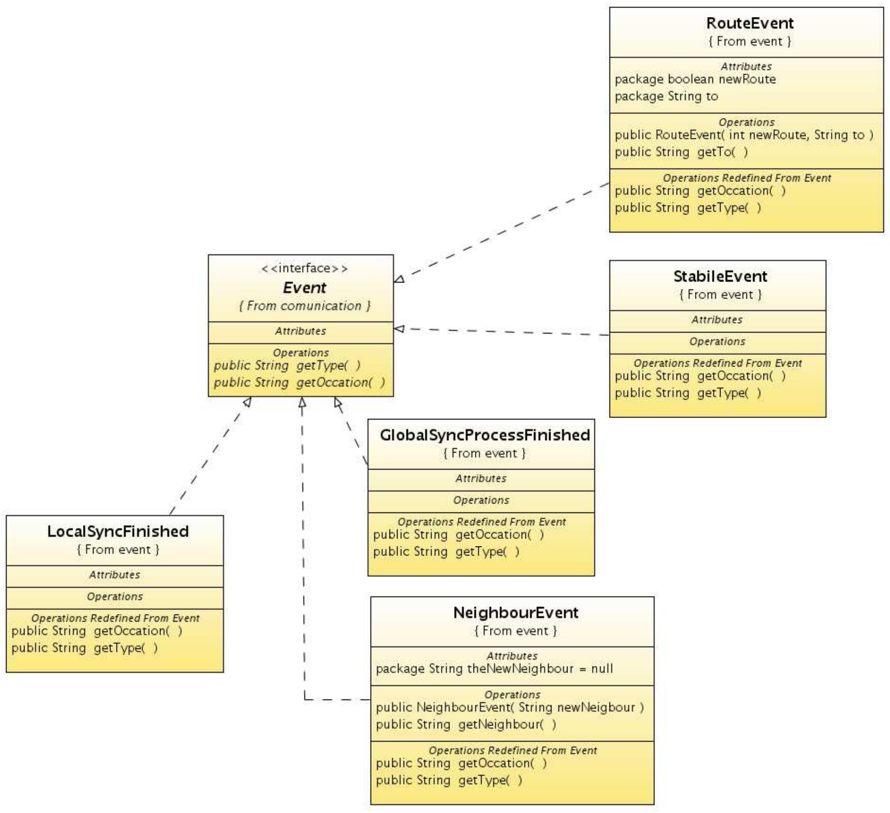

RouteEvent
{ From event }
Attributes
package boolean newRoute
package String to
Operations
public RouteEvent(int newRoute, String to)
public String getTo( )
Operations Redefined From Event
public String getOccation( )
public String getType( )
<<interface>>
Event
{ From comunication }
Attributes
Operations
public String getType( )
public String getOccation( )
StabileEvent
{ From event }
Attributes
Operations
Operations Redefined From Event
public String getOccation( )
public String getType( )
GlobalSyncProcessFinished
{ From event }
Attributes
Operations
Operations Redefined From Event
public String getOccation( )
public String getType( )
LocalSyncFinished
{ From event }
Attributes
Operations
Operations Redefined From Event
public String getOccation( )
public String getType( )
NeighbourEvent
{ From event }
Attributes
package String theNewNeighbour = null
Operations
public NeighbourEvent(String newNeigbour)
public String getNeighbour( )
Operations Redefined From Event
public String getOccation( )
public String getType( )

Figure D.3: The UML model showing the DENS event model.

Appendix E

Re-execution of the experiments

To do a re-execution of the experiments one needs to follow some steps;

1. Setup a NEMAN server. This involves getting the source code and kernel paches and applying the patch and compiling the NEMAN suite. This can be acomplished by contacting the authors of [13].

2. Once the emulation enviroment is up and running on the server with the wanted number of taps, the experiments need to be setup.

   This is done by starting *iemul*, from the NEMAN suite, setting it up with the correct server address.

   loading the correct scenario

   starting the control packet tunneling, *tunnel.pl*

   moving the *test-setup* files to the server. Together with the jar file, and linking the *r*un.sh script with the correct jar file.

3. starting the experiments by issuing *run_tests.sh*, waiting for it to start all nodes and logging.

4. pressing the play button in the *iemul* GUI, running the tests and pressing "any-key" in the server terminal to mark the finishing of the test run, which results in a tar ball being produced with all the logs.