Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Adrian Friedli

A Lightweight Data Dissemination Algorithm for Multi-hop Wireless Networks

Semester Thesis SA-2012-09

Tutor: Mahdi Asadpour
Co-Tutor: Dr. Karin Anna Hummel
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

In a mesh network consisting of multiple nodes, data may be sent either through a routing mechanism or through some other way. One of the other methods of data transmission is data dissemination, where each node is able to send data to every other node in the mesh network and each node decides to forward the data using some algorithm. For this semester thesis a lightweight data dissemination algorithm has been implemented and validated as part of the SWARMIX project.

Contents

Chapter 1

Introduction

1.1 Motivation

At places where rescue missions are needed, most often no comminucation infrastructure is available, maybe due to their absence, due to destruction of equipment or due to unavailability of elecricity. Rescue organisations need communication possibilities within their teams or with teams of other organisations. The SWARMIX project can help here. Network nodes in form of unmannned aerial vehicles may get sent out to build some kind of mesh network.
In such a mesh network, each node is able to communicate directly with it's neighbours but not with the rest of the network. Nodes may act as a relay for other nodes. With the help of multiple relays any node should be able to reach all the other nodes in the network.
Until now, SWARMIX nodes used a routing protocol over the mesh network to communicate with each other. Such a routing protocol is good for one-to-one communication in the network. However for sending data like GPS information from each node to all the other nodes in the network such a routing protocol is not desirable. Hence, a new protocol is needed for this one-to-many communication.
A simple solution would be to broadcast such data as packets to all neighbors and also broadcast each received network packet to all neighbors while keeping track of already sent packets to avoid sending the same packet twice or more and thus creating a loop. This approach is called simple flooding or blind flooding. However, most wireless network applications operate on a shared medium where only one user can transmit data at the same time within a certain area. With this simple flooding the use of the shared medium gets saturated quickly. In order to optimize the network load a better aproach is needed. Each node should decide in a distributed manner if it should re-broadcast a received packet.

1.2 The Task

The tasks of this thesis include the familiarization with the topic, the implemention of a data dissemination algorithm and the testing of said algorithm in a real-life scenario. The three tasks can be described as follows:

Familiarization: The student familiarizes himself with the topic by reading literature about data dissemination, by evaluating libraries providing network functionality and by setting up wireless network hardware.

Implementation: The student implements a data dissemination algorithm in the C++ programming language, bearing in mind the algorithm should be both capable of running on low-end embedded systems and being accessible in form of a library. Additionally documentation is done using Doxygen.

Test: After validating that the implemented algorithm works in a simple simulator, real-life testing is done using several nodes with wireless hardware. Multiple metrics are observed under different configuration parameters of the algorithm and are compared to each other.

1.3 Related Work

An important relevant work was [1], which heavily influenced the implemented algorithms and defined the measurement metrics which have been used in this thesis to analyze and compare the algorithms. Another important work was [3], which also influenced our one- and two-hop algorithms. The work [5] introduces storage to the network, similar to our store and forward extension. Another work we looked at was [4]. Their algorithm not suited to our needs, because it relies on the knowledge of the position data.
We also looked at spray and wait [2], a delay-tolerant algorithm, which sends out a limited amount of copies of a message. Messages then get distributed by the help of the node's mobility. We didn't make use of this because we didn't consider mobility.

1.4 Overview

After a short description of the goals and problems of this semester thesis in chapter 2, the implemented algorithms as well as the software design get explained in chapter 3. Chapter 4 introduces the testing setup, the used simulator and the measurement metrics and presents the results of the measurements. Chapter 5 presents ideas to build upon the work of this thesis. And finally, chapter 6 concludes the thesis with a short summary.

Chapter 2

Goals and Problems

2.1 Algorithm

In a broadcast environment network congestion occurs when too many packets are being transmitted at the same time, this will result in a low total data transfer rate. Simple flooding is rather inefficient and better solutions exist, there exist multi-hop algorighms, location-based algorithms, delay-tolerant algorithms and more.

2.2 Implementation

Prior to this thesis no data dissemination implementation exists for the Swarmix project. The goal of this semester thesis is to implement an improved flooding algorithm that can be used in real world environments. The resulting application is to be able to run on the Gumstix hardware running the GNU/Linux operating system. Care has to be taken in the choosing of libraries and it has to be kept in mind that the gumstix hardware is limited in respect to system resources. Also the implemented algorithm should not be too complex.

2.3 Measurement

Furthermore the resulting application has to be tested and validated, and some measurements have to be performed in a real world environment.

Chapter 3

Software design

3.1 Algorithms

In this thesis, three different algorithms have been implemented and analyzed. The implemented algorithms were inspired by algorithms described in [1]. All algorithms work by broadcasting packets, so that every neighbor of the sending node will receive the packet with a high probability. Each packet contains the node id of the originating node and a sequential packet id. This tuple is a unique identifier and will be used to detect already received packets (see also Table 3.2).

3.1.1 Simple flooding

In simple flooding, every node forwards every packet it receives unless it has already forwarded that packet.

3.1.2 1-Hop algorithm

The implemented 1-hop algorithm works as follows. Every node sends out hello packets, which only contain the senders node id. With the help of these hello packets each node maintains a list of neighbors. The hello packets can be omitted if enough other packets are being transmitted.
Whenever a node transmits a data packet, be it an initial transmission or a forwarded packet, it includes its list of neighbors in the packet header.
Whenever a node receives a data packet, it puts that packet into a queue and sets a timer to a random timeout. While waiting for the timeout, the node may receive the same packet again from another neighbor. Whenever that happens, the node combines the neighborlist included in the queued packet with the neighborlist included in the newly received packet to form a list of nodes which may already have received the packet. When the timer runs out, the node compares its own neighborlist with the list of nodes generated while waiting for the timeout and only forwards the packet, if the node has any neighbors which are not in the received list.
The random timeout is needed to generate this received list, it also helps to reduce collisions because not all the neighbors will forward a packet at the exact same time.
As an extension to the 1-hop algorithm, a store and forward functionality was implemented. Using this functionality, each node can keep a packet for some time and then send it to a node newly joining the set.

3.1.3 2-Hop algorithm

The implemented 2-hop algorithm works as follows. Every node sends out hello packets, which contain both its sender's node id and a list of the sender's neighbors. Using this list, each node can then maintain a list of its two-hop neighbors, i.e. its neighbor's neighbors. As above, the hello packets can be omitted if enough other packets are being transmitted.
In this algorithm, the decision of whether to forward a packet is not made by each node by itself, but by each forwarding or sending node. Every time a node transmits a packet it decides which

libnet	
+	-
easy to use	
easy portable to DOS/windows	
multiple backend drivers	
	no broadcast
	no support for IPv6
	quite old (last update from 2003)

POSIX functions	
+	-
more flexible, no limitations from a library	
easy to extend to ipv6	
no dependencies on third party libraries	
	complex usage
	greater amount of work to port to other systems

Table 3.1: comparison of the network libraries

of its neighbors should forward the packet. Whenever a packet is being sent, be it an initial transmission or a forwarding, a node looks at the list of its neighbors and two-hop neighbors and chooses the smallest set of neighbors with which all two-hop neighbors can be reached. This list is then included in the packet header as the forwarders list. Each neighboring node will only forward a packet if it is included in the forwarders list.

3.2 Implementation

According to the task description, the implementation should be written in the C and/or the C++ programming language. Because C++ allows an object oriented design, which can lead to a cleaner implementation and because the student likes to improve his programming skills in this language, C++ was chosen.
The C++ programming language is available for many platforms, with a variety of compilers. The GNU Compiler Collection (GCC), as found on every GNU/Linux distributions, was used.

3.2.1 Evaluation of a network library

The program to implement for this thesis should run on the GNU/Linux operating system. On GNU/Linux there are basically two ways you can implement network access in a program. Either you can use pure POSIX functions for your program or you can use an existing network library, which calls the POSIX functions for you and presents a simpler and more abstract interface. We have compared libnet against native POSIX functions. The results of this evaluation are shown in table 3.1.
The task does not require multiple network protocols, it does not require the software to run on Windows nor is IPv6 networking a requirement. But the task requires to be able to send broadcast packets in the network, which is not provided by libnet. We could extend the libnet library to do so, but we would still not gain much against using POSIX functions directly.
Because of the above mentioned criterias about libnet versus using POSIX functions, we decided to use the latter and encapsulate the POSIX network functions in a C++ object.

3.2.2 Implementation of the networking interface

The networking interface consists of three classes.

The network class provides the functionality for sending and receiving packets and efficiently polling sockets and other file descriptors.

The packet class represents a network packet with source and destination address and a data buffer for the payload

The address class represents a network address

The collaboration graph of these three classes can be seen in figure 3.1.

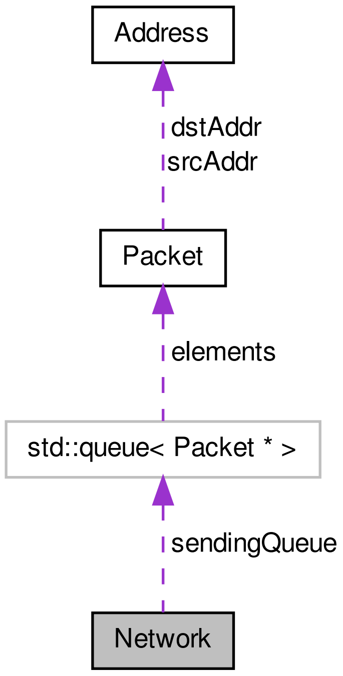

Figure 3.1: Collaboration graph of the network classes

3.2.3 The network class

The `Network` class is the core class of the network functionality in the program written for this thesis. It represents the UNIX network functions `sendto` and `recvfrom` for sending and receiving UDP packets, as well as `poll` for efficiently handling non-blocking operations, in an object oriented way.
It allows non-blocking sending and receiving of network packets. For non-blocking sending of a packet when the operating system would block the operation, a sending queue is used.
A user of this networking interface should have a single instance of the `Network` object. The important functions this network object provides are:

bindSocket binds the network socket to a given UDP port

receive receives a packet from the network and either creates a packet object or fills a provided one

queuePacket adds a packet to the sending queue

pollQueue polls the network socket, sends out packets in the sending queue when the operation won't block, returns either when there is a packet to receive or when a given timeout is reached

3.2.4 The monotonic time class

A `MonotonicTime` class instance represents a point in time. The value of a single object is arbitrary and can only be used for comparing relative time differences to other objects of this class. It is independent of the system time and thus is not affected by changes to the system time. The static method `MonotonicTime::now` returns a new object representing the point in time the function was called. A multitude of arithmetic operations are implemented in the class. For instance calculating time difference of two objects or adding a time difference to an object and thus creating a new object. And of course comparison operators are iplemented as well.

3.2.5 The event dispatcher class

For handling various types of events, an event dispatcher must be used. This is done by inheriting from the abstract class `EventDispatcher` and implementing the method `dispatch`. The event dispatcher then can be registered for use at various places.

3.2.6 The dissemnination class

The dissemination class is the main class of this project. A user of this class should have a single object as an instance of this class. For this object to work properly, at least one receiver object has to be registered and some configuration has to be done.
Upon instantiation of the dissemination object, required configuration parameters have to be given as arguments to the constructor. These include the string of the network interface to use, the UDP port number to listen on, which should be the same on all nodes, and the node id of this particular node, which should be unique across all nodes. Optional configuration may be done by setting members of the dissemination object, otherwise default values will be usued.
A collaboration graph including almost all classes can be seen in Figure A.1 in the appendix.
A user of the dissemination class has to implement a packet receiver. This is done by inheriting from the abstract class `PacketReceivedEventDispatcher` and implementing the method `dispatch`, similar to the event dispatcher class. The packet receiver has then to be registered by setting the member `packetReceiver` pointer to the packet receiver object. The object's dispatch function is then called with a packet object and the sending node as arguments by the dissemination object whenever a new packet was received.
Using the `addTimerEvent` method custom event dispatchers can be added and will get dispatched after a custom timeout. The timer events can be configured as single or repeated events. They can be safely added when the dissemination object is already running.
After configuration and registering of the packet receiver the `run` method must be called, which will never return.

3.2.7 Application specific packet types

Beside of the `ArbitraryDataPacket` class, custom data packet classes can be used. To accomplish that, one has to make a class inheriting from the abstract `DataPacket` class and implement the `serialize` function. The `serialize` function takes a pointer to a buffer as an argument, it should write the object's content to that buffer and it should return the buffer's length. For deserialization the classes constructor may take a buffer pointer and a buffer length as arguments and should fill the object's members with the data from that buffer.
Two classes for application specific data have been implemented so far, `PositionInformation` and `TopologyInformation`. `PositionInformation` objects may be used to transmit GPS position data including latitude, longitude, altitude, direction, velocity and a timestamp. `TopologyInformation` objects may be used to transmit neighborhood information containing a list of 2-tuples describing neighbor relations. The latter one also gets used by the optional store and forward mode of the implemented algorithms.

3.2.8 Packet format

All our packets get transmitted as UDP packets. Inside the UDP packet payload an own data packet header gets included before the actual payload data. Table 3.2 shows the format of

Bit offset	Field description	
0	forwarding node id	
8		
16	originating node id	identifies packet uniquely
24		
32	packet id	
40		
48	payload type	
56	neighbor list length	
64	neighbor list	
⋮		
f	forwarder list length	
f + 8	forwarder list	
⋮		
p	payload	
⋮		

Table 3.2: data packet format

such a UDP payload. For simplicity both algorithms use the same packet format and the 1-hop algorithm always includes an empty forwarder list of length zero.

3.3 Simulator

For code testing and validation, a simple simulater has been implemented.
The simulator tests *the same code* as will be run on the target node. The simulator does not simulate packet loss due to congestion and also does not simulate latency. The simulator allows for fast development cycles because all the testing node programs can be run on the same host. The simulator gets started on a host and then listens on a specified port. The node programs using the dissemination algorithm must then be reconfigured to send all outgoing packets to the simulator port instead of the broadcast address on the wireless interface. The node programs must also be reconfigured to listen on different ports, so they can be run all on the same host.

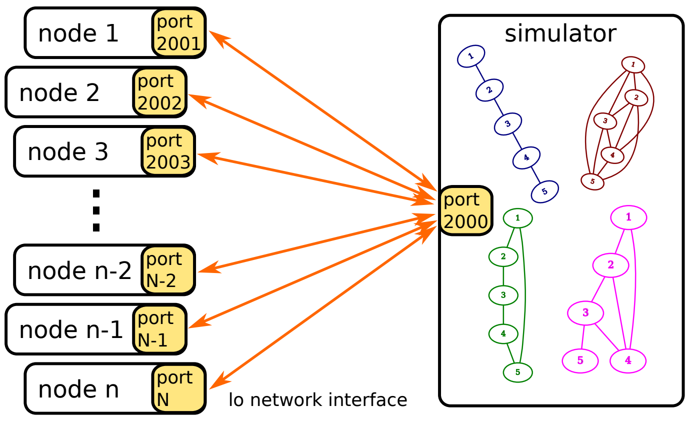

Figure 3.2: Overview of the simple simulator

Chapter 4

Results

4.1 Measurement metrics

In order to analyze the implemented algorithms, the metrics defined by Ni et al. in “The broadcast storm problem in a mobile ad hoc network”[1] have been used.

Reachability

$$r = \frac{\#_{\text{received}}}{\#_{\text{nodes}} - 1} \tag{4.1}$$

Saved rebroadcast ratio

$$f = \frac{\#_{\text{received}} - \#_{\text{forwarded}}}{\#_{\text{received}}} \tag{4.2}$$

Average latency

$$a = \frac{1}{\#_{\text{receivers}}} \sum_{\text{receivers}} \text{latency} \tag{4.3}$$

Reachability tells us the average ratio of nodes that receive a packet by any node. The saved rebroadcast ratio describes how many forwardings have been saved compared to simple flooding. And average latency tells us the latency of a packet disseminated in the network taking the mean of all nodes.

4.2 Measurement setup

It was difficult to set up test nodes in a real environment such that some nodes connected to each other and others are not. Therefore, it was decided to use a virtual topology instead. In that virtual topology five laptops, each equipped with an 802.11n USB Wifi dongle in adhoc mode in the 5 GHz band, were used as nodes. All laptops were put in the same room. To simulate unconnected nodes iptables was used to setup a firewall that drops incoming packets from “not connected” nodes on each laptop.

4.2.1 Virtual topologies

The implemented algorithms were tested in four different virtual topologies. The used topologies can be seen in Figures 4.1, 4.2, 4.3 and 4.4. These different topologies have been chosen to test the algorithms with different connectivity degrees between the nodes.

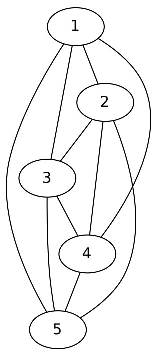

Figure 4.1: The 'all connected' topology

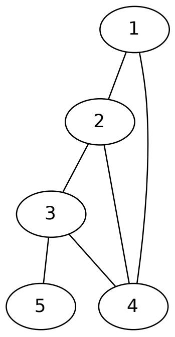

Figure 4.2: The 'some connected' topology

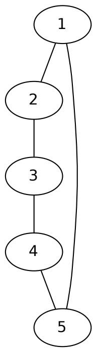

Figure 4.3: The 'circle' topology

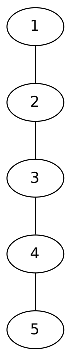

Figure 4.4: The 'line' topology

4.3 Measurement results

For the reachability distribution graphs (Figures 4.5, 4.7, 4.9 and 4.11), darker colors mean lower reachability. The darkest color corresponds to 0 and the brightest to 1; the colors between correspond each to 0.25, 0.5 and 0.75. The saved rebroadcast ratio graphs (Figures 4.13, 4.14, 4.15 and 4.16) each show the mean with standard deviation. The average latencies are the only measurements providing continuous values. Therefore, box plots have only been used for those plots (Figures 4.17 to 4.28).

4.3.1 Graphical results

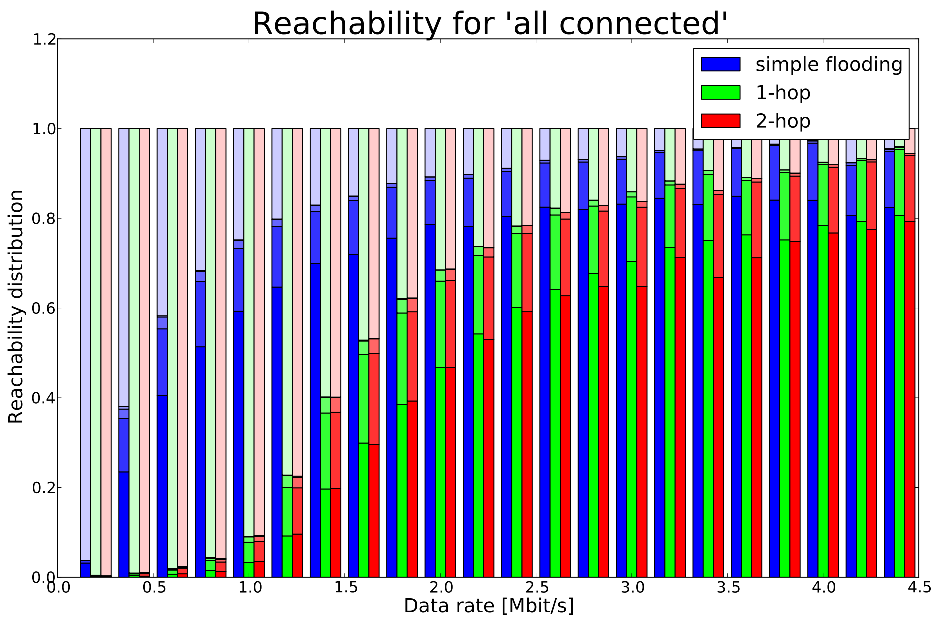

Figure 4.5: Reachability distribution for the ’all connected’ topology

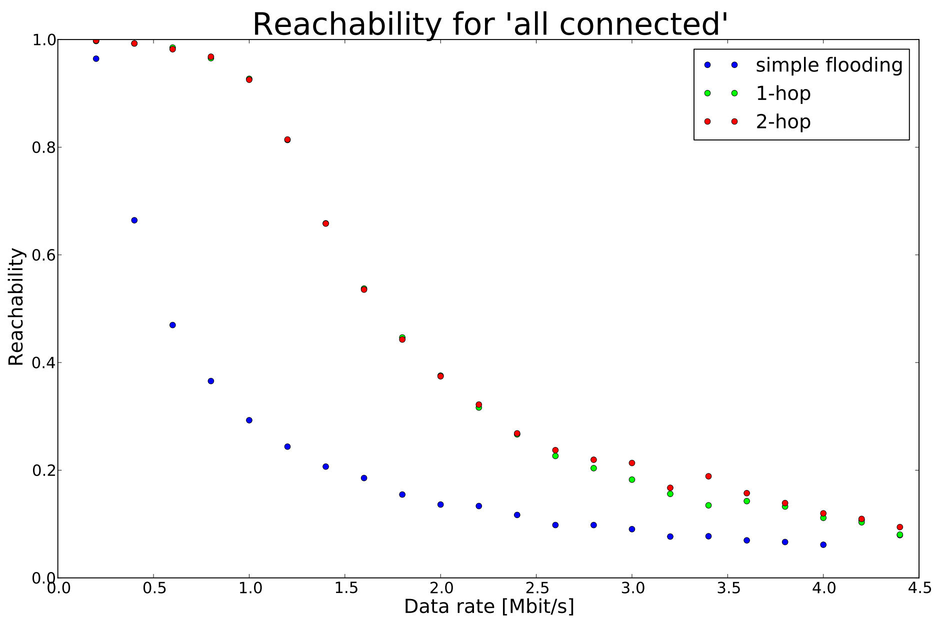

Figure 4.6: Reachability for the ’all connected’ topology

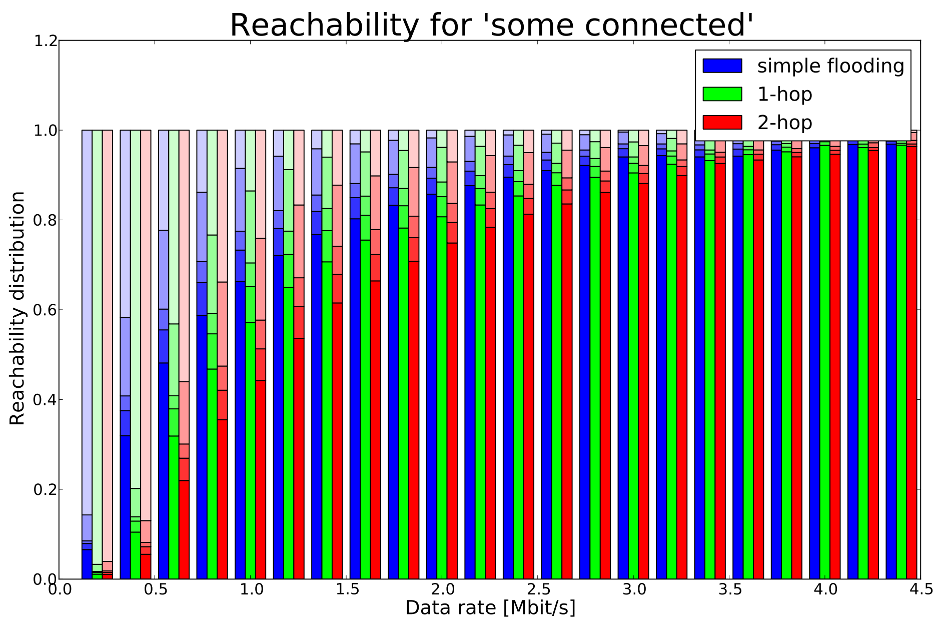

Figure 4.7: Reachability distribution for the ’some connected’ topology

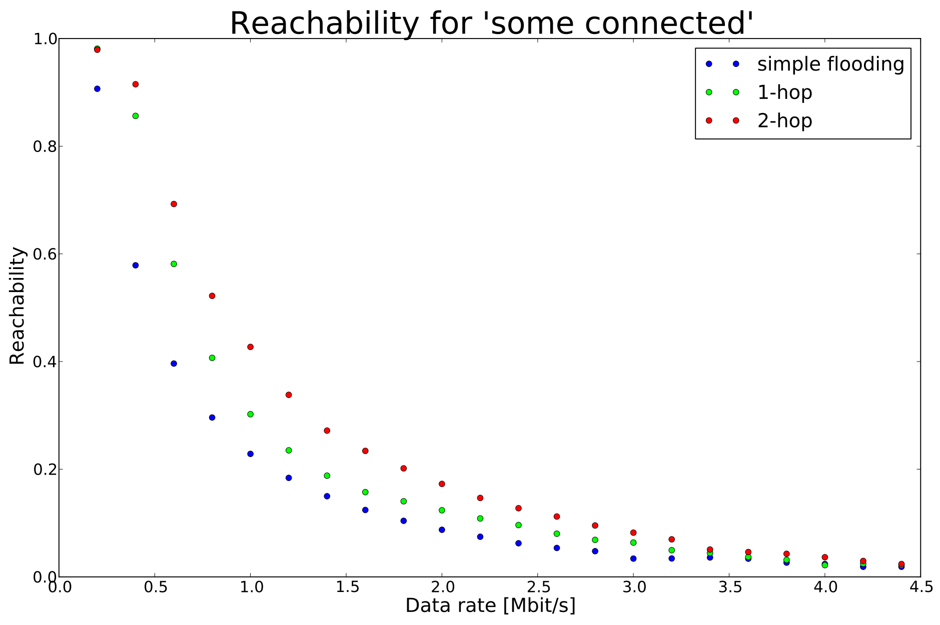

Figure 4.8: Reachability for the ’some connected’ topology

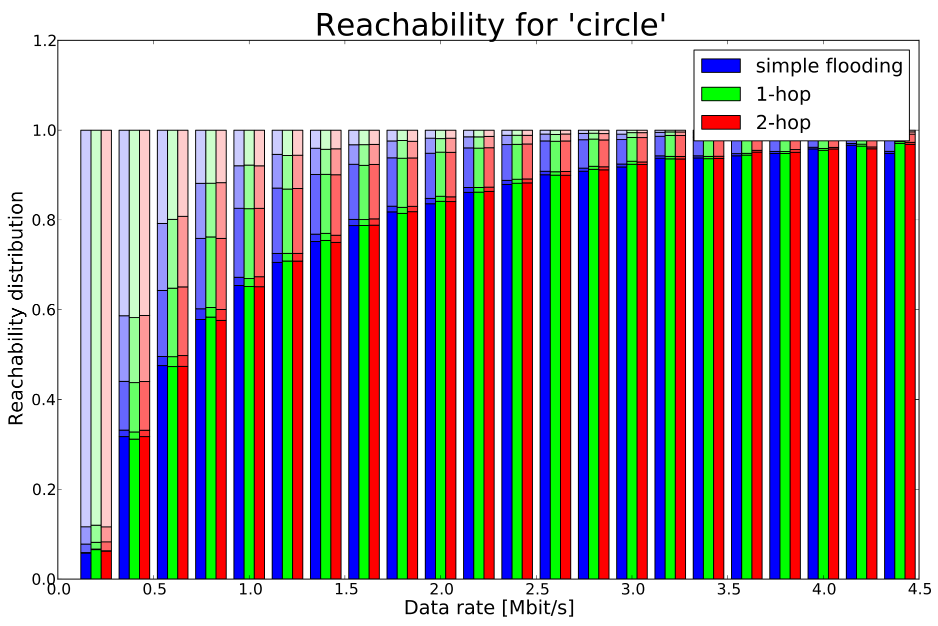

Figure 4.9: Reachability distribution for the 'circle' topology

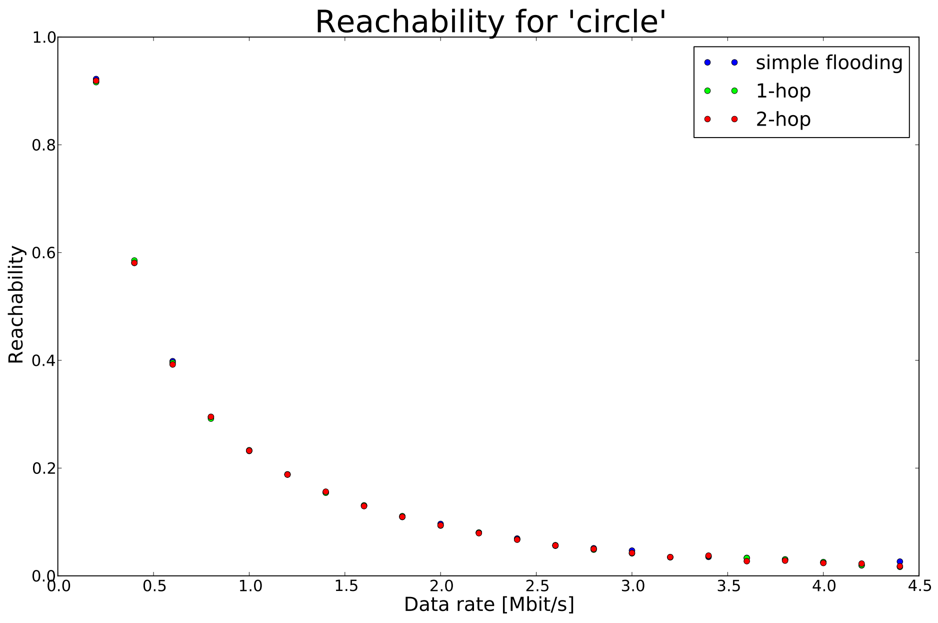

Figure 4.10: Reachability for the 'circle' topology

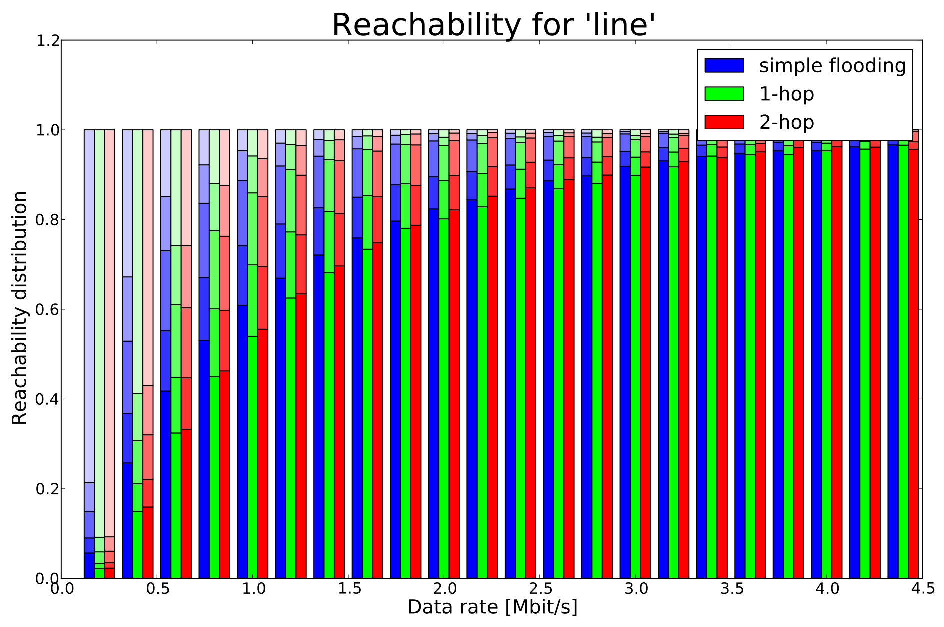

Figure 4.11: Reachability distribution for the ’line’ topology

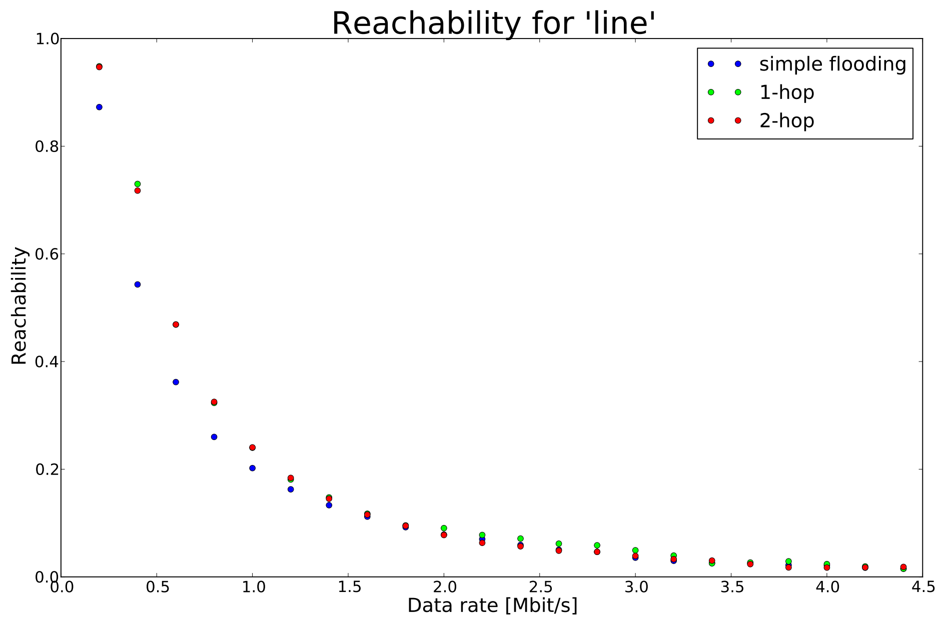

Figure 4.12: Reachability for the ’line’ topology

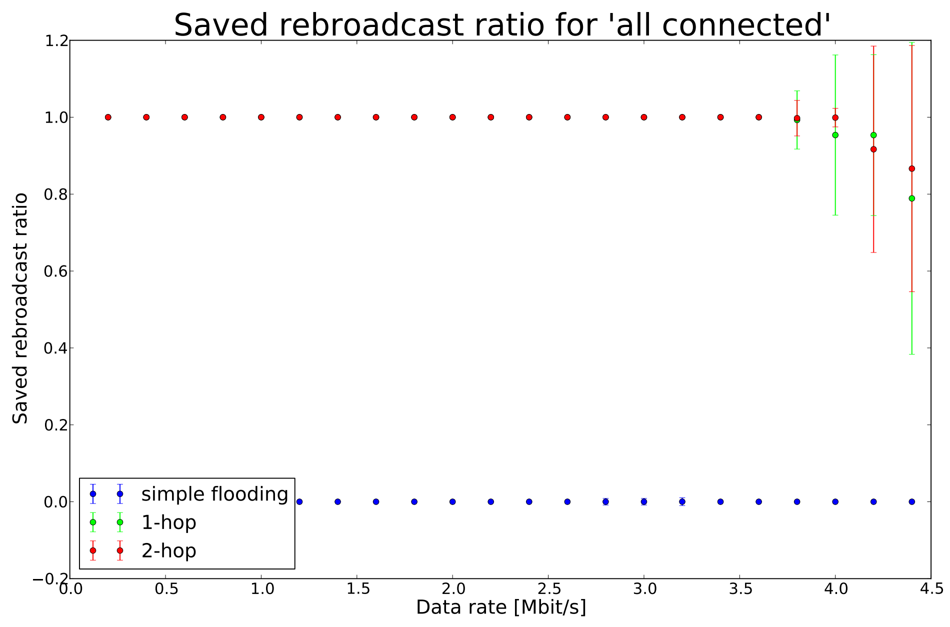

Figure 4.13: Saved rebroadcast ratio for the ’all connected’ topology

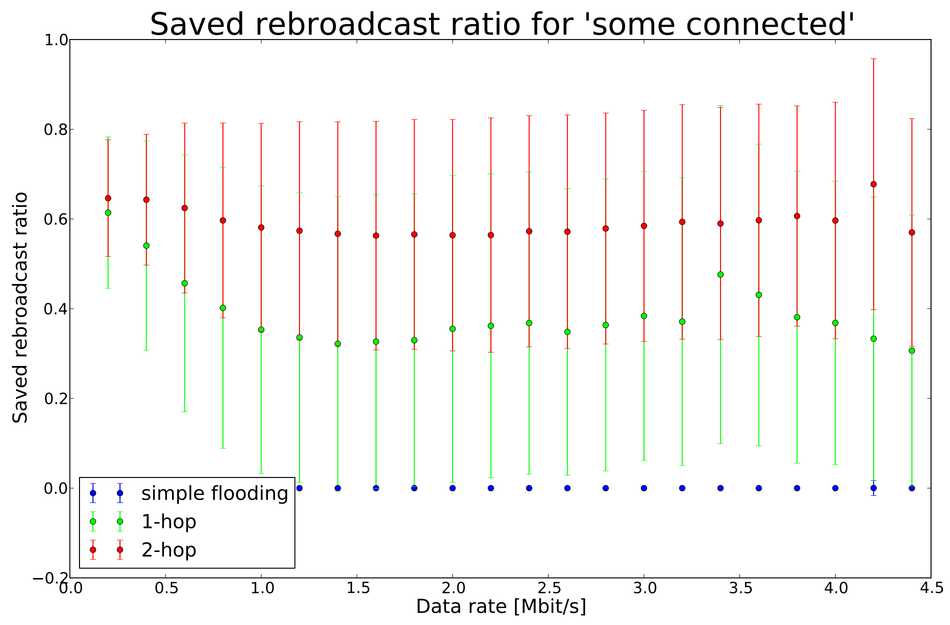

Figure 4.14: Saved rebroadcast ratio for the ’some connected’ topology

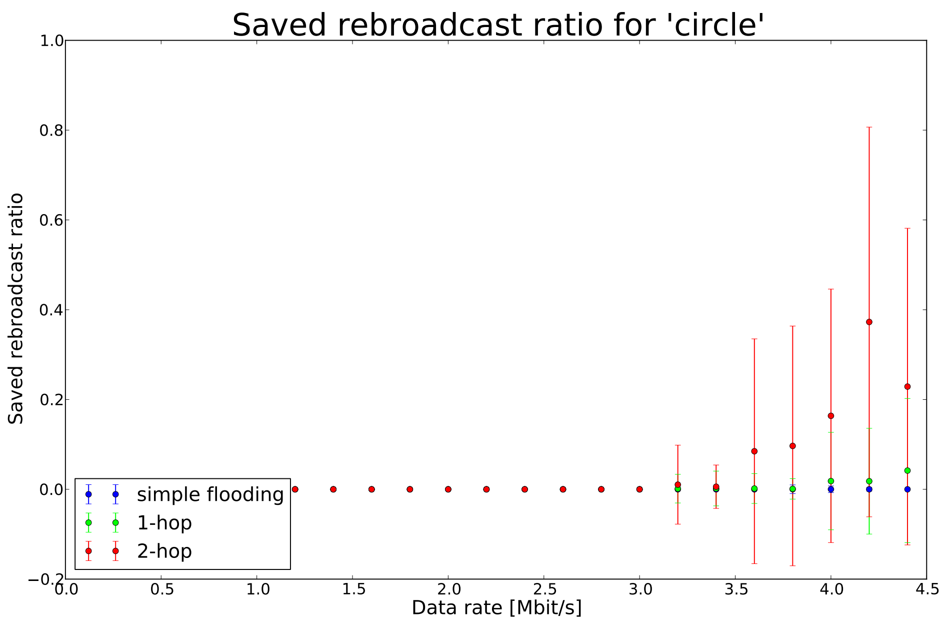

Figure 4.15: Saved rebroadcast ratio for the ’circle’ topology

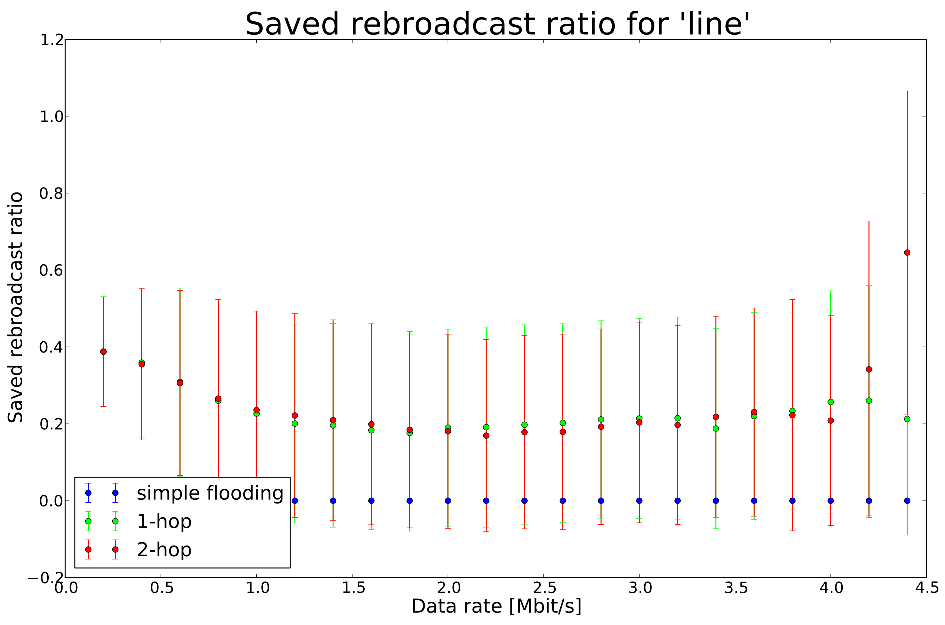

Figure 4.16: Saved rebroadcast ratio for the ’line’ topology

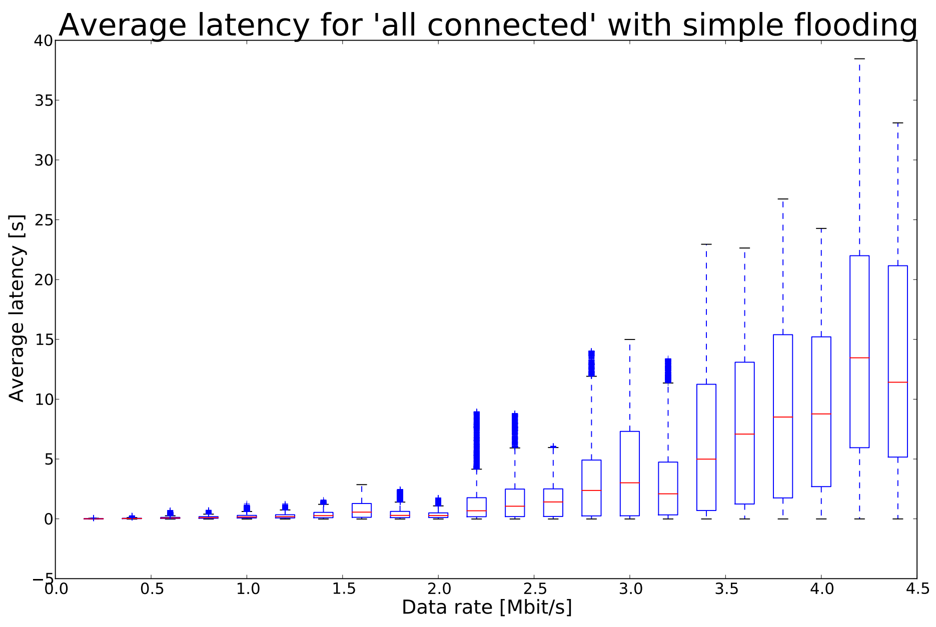

Figure 4.17: Average latency for the ’all connected’ topology with the simple flooding algorithm

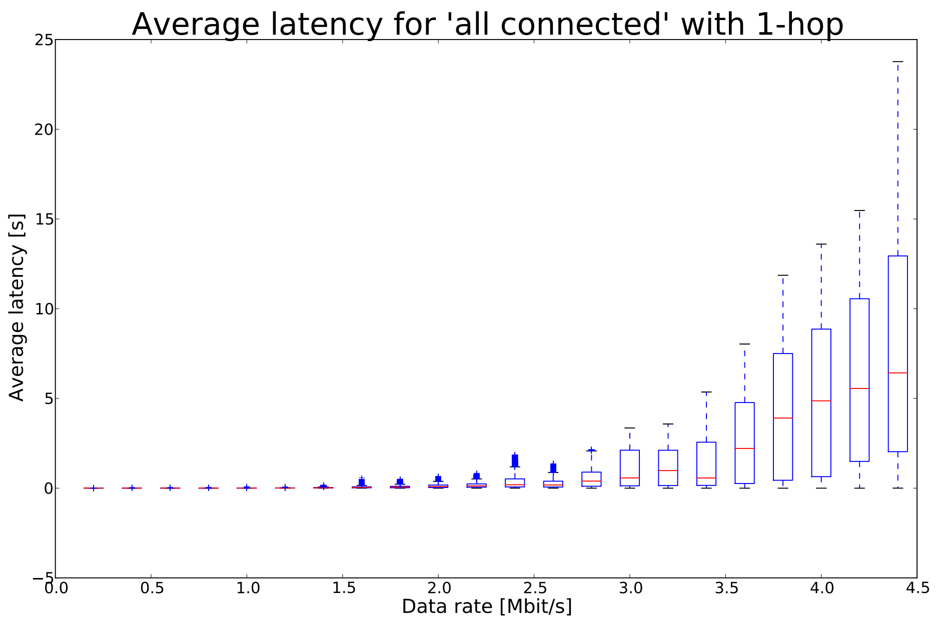

Figure 4.18: Average latency for the ’all connected’ topology with the 1-hop algorithm

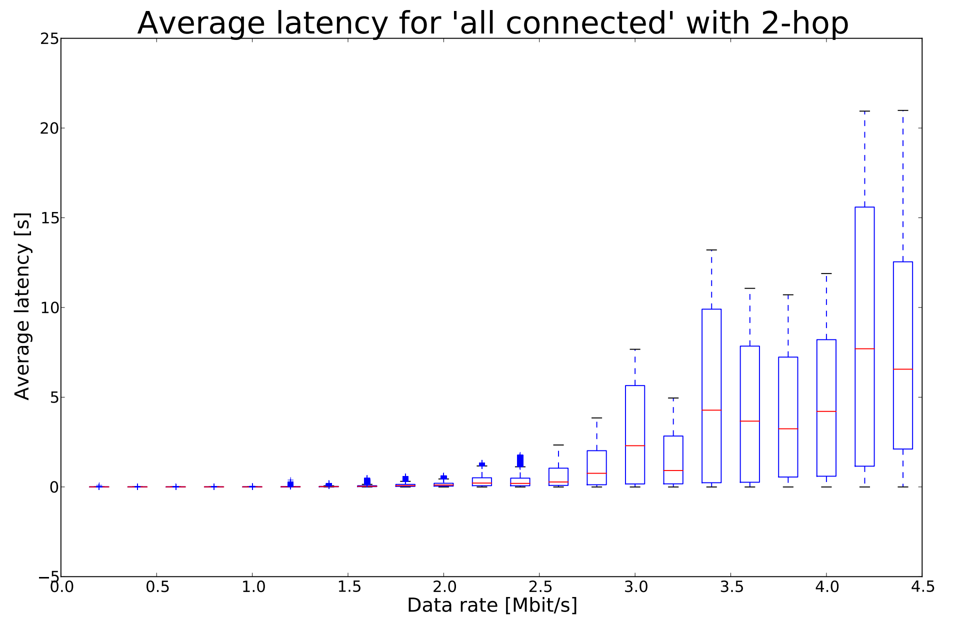

Figure 4.19: Average latency for the ’all connected’ topology with the 2-hop algorithm

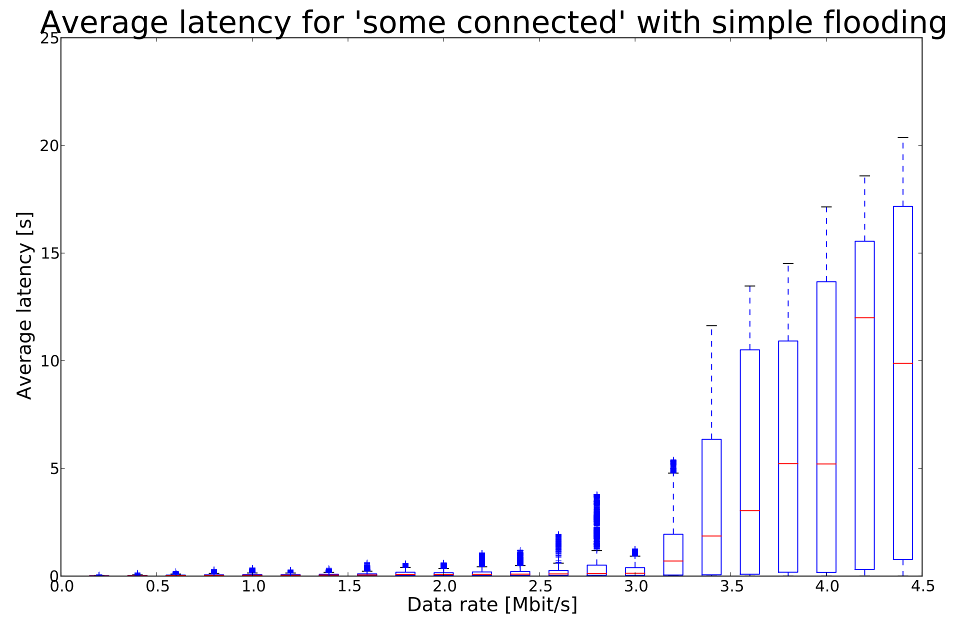

Figure 4.20: Average latency for the ’some connected’ topology with the simple flooding algorithm

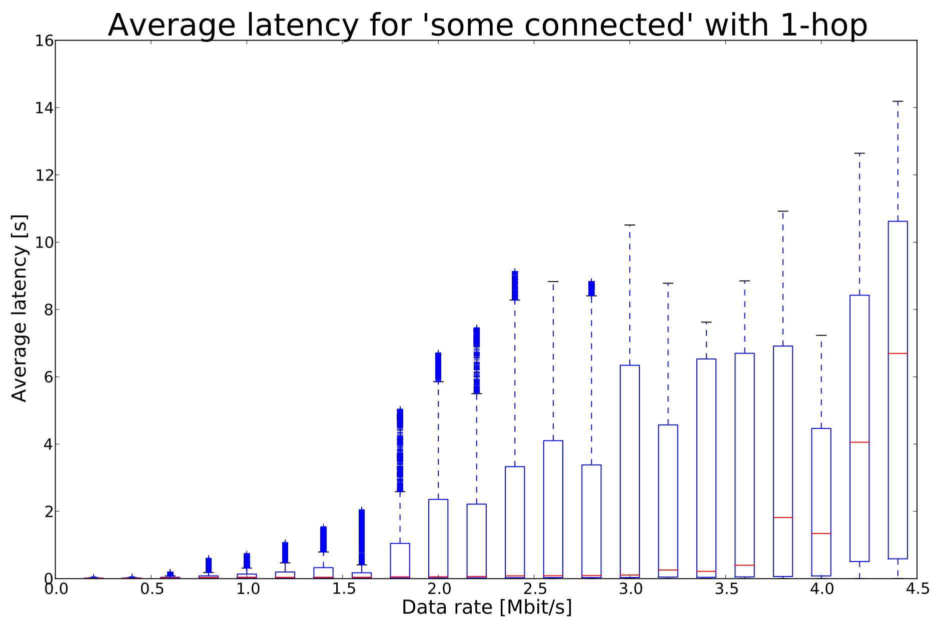

Figure 4.21: Average latency for the ’some connected’ topology with the 1-hop algorithm

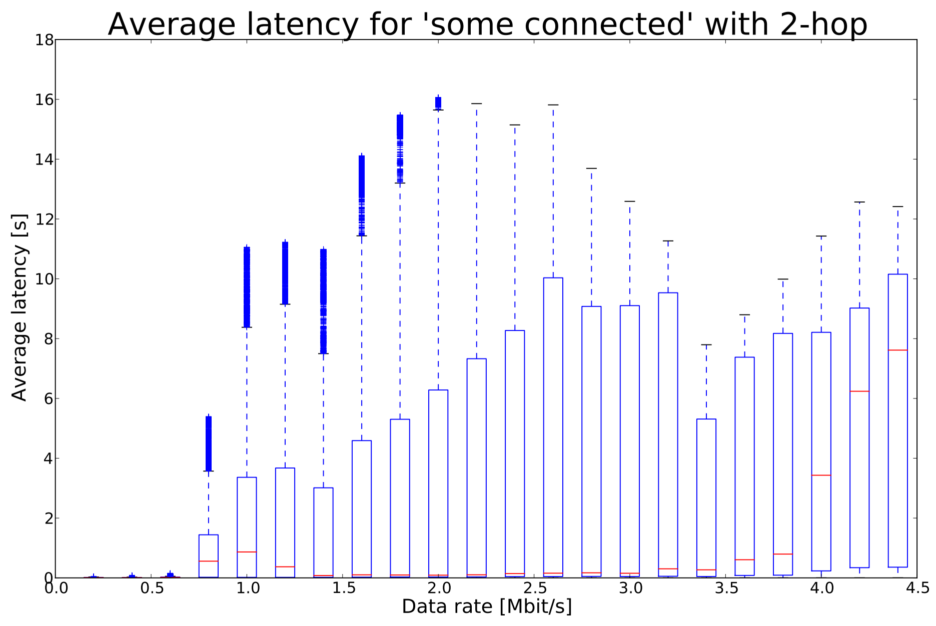

Figure 4.22: Average latency for the ’some connected’ topology with the 2-hop algorithm

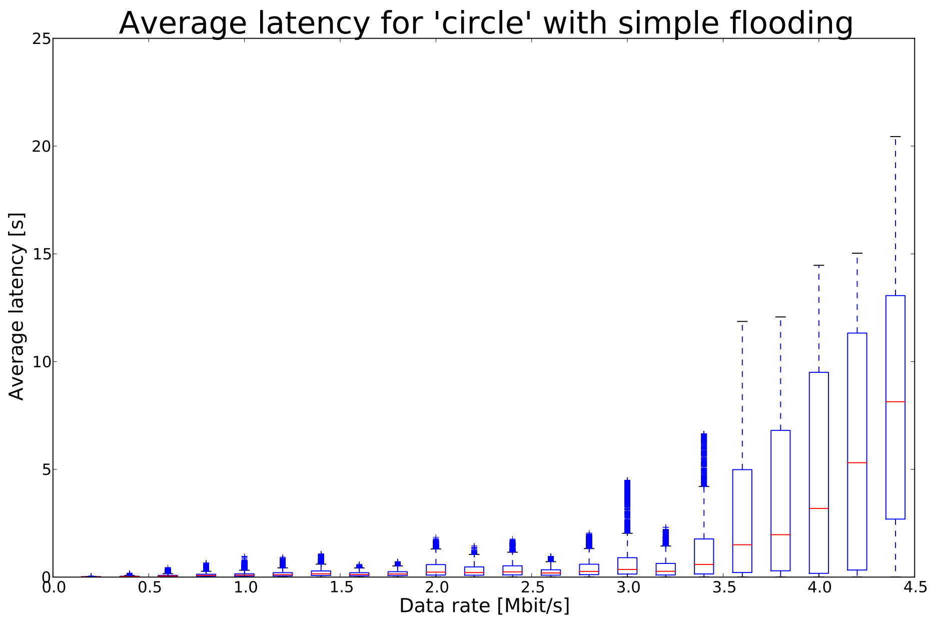

Figure 4.23: Average latency for the ’circle’ topology with the simple flooding algorithm

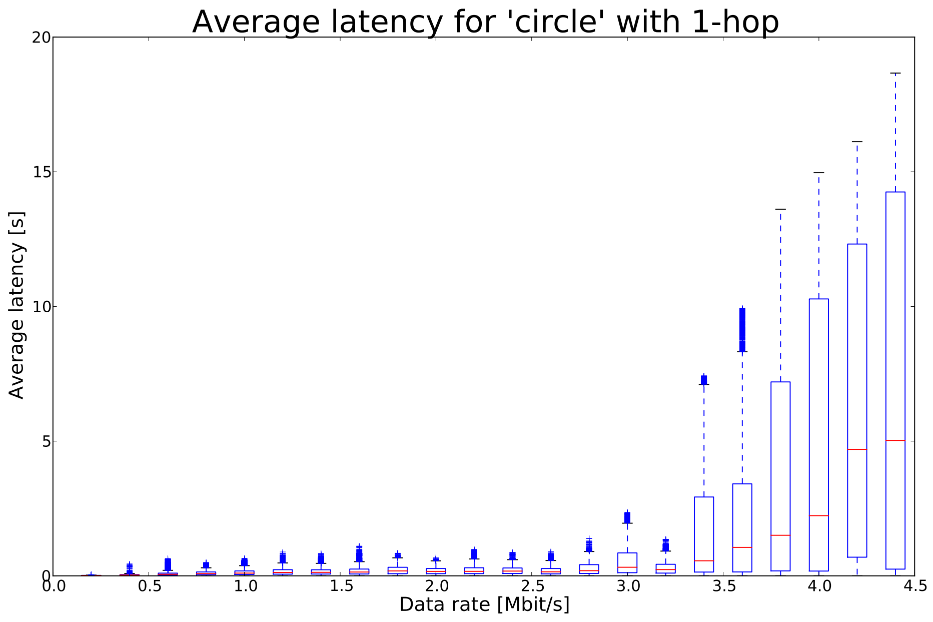

Figure 4.24: Average latency for the ’circle’ topology with the 1-hop algorithm

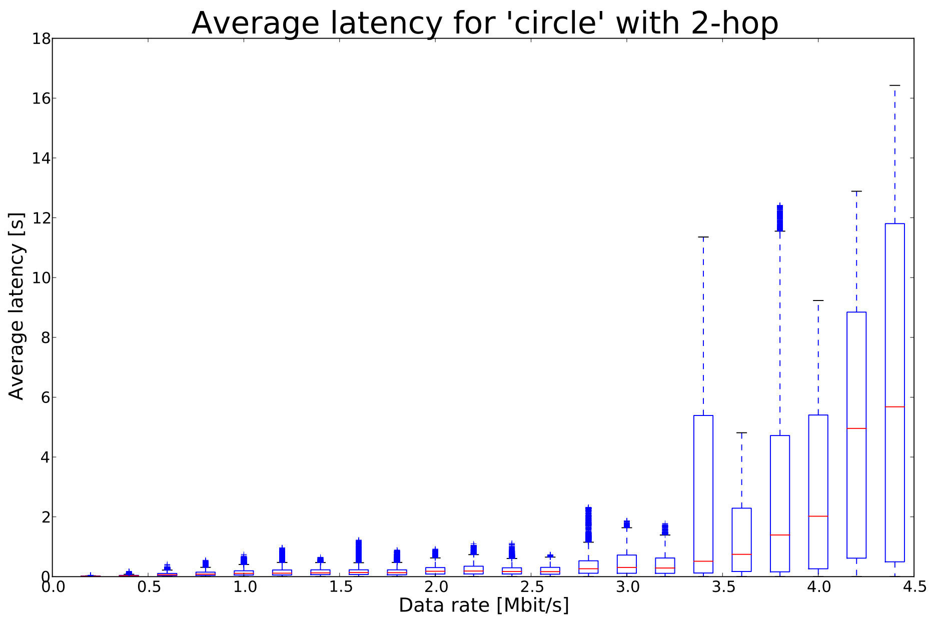

Figure 4.25: Average latency for the ’circle’ topology with the 2-hop algorithm

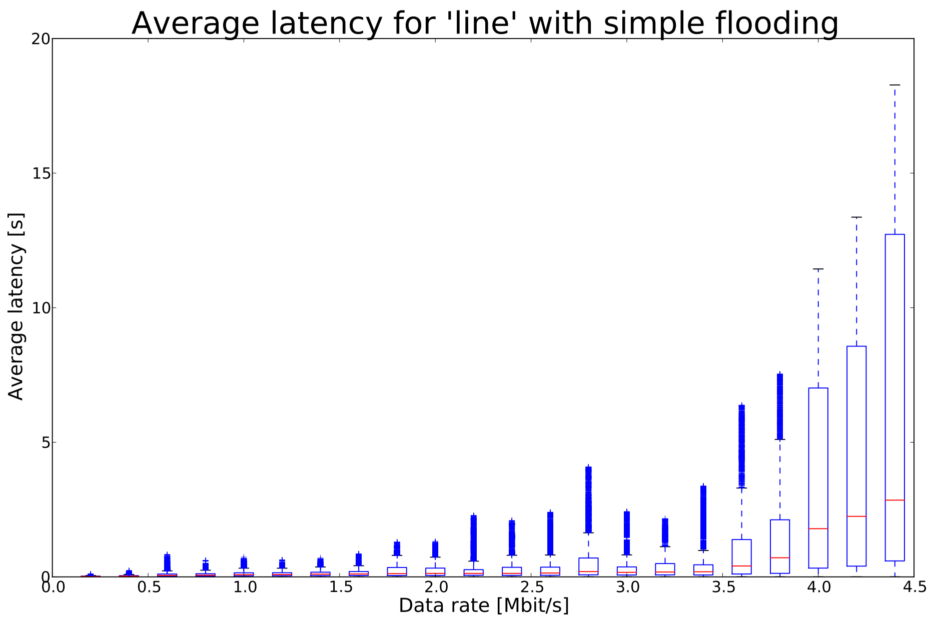

Figure 4.26: Average latency for the ’line’ topology with the simple flooding algorithm

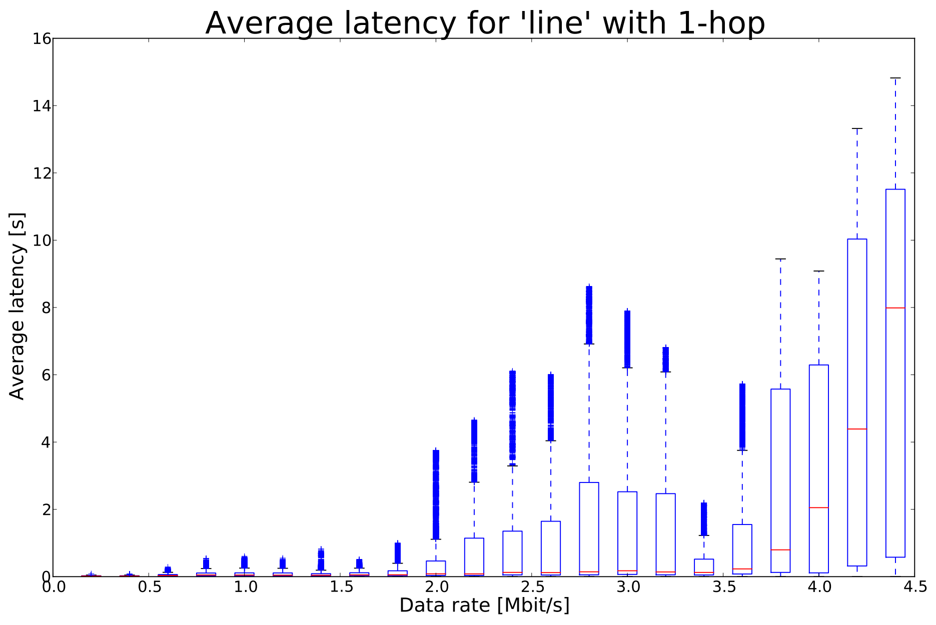

Figure 4.27: Average latency for the ’line’ topology with the 1-hop algorithm

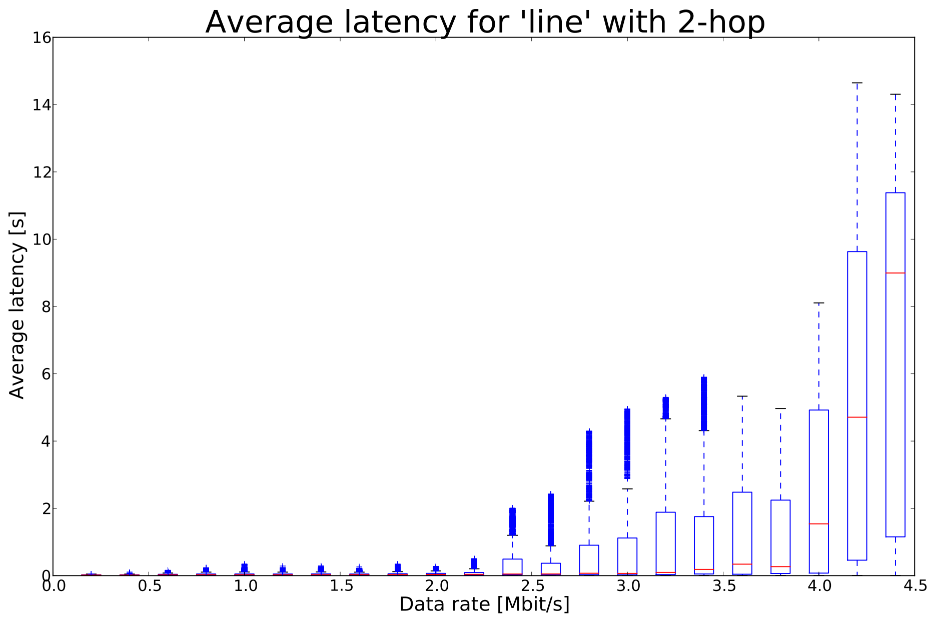

Figure 4.28: Average latency for the ’line’ topology with the 2-hop algorithm

4.3.2 Discussion

'all connected' topology

Figure 4.5 and Figure 4.6 show the reachability distribution and the mean reachability, respectively, of the measurements in the 'all connected' topology for different data rates. It is clearly visible, that in this topology the two algorithms 1-hop and 2-hop perform about equally well and substantially better than just simple flooding. The reason for that huge performance difference can also be seen by looking at Figure 4.13, depicting the saved rebroadcast ratios. Disregarding the noise at higher data rates the two algorithms 1-hop and 2-hop rebroadcast nearly no packets and simple flooding just rebroadcasts every packet it receives. The Figures 4.17, 4.18 and 4.19 depict average latencies at different data rates for each algorithm. Again, we can see, that the two algorithms 1-hop and 2-hop perform better than simple flooding.

'some connected' topology

Figure 4.7 and Figure 4.8 show the reachability distribution and the mean reachability, respectively, of the measurements in the 'some connected' topology for different data rates. The 2-hop algorithm performs best and the 1-hop algorithm is between the 2-hop algorithm and simple flooding. Also by looking at 4.14 one can see that, in this topology, the 2-hop algorithm can save a significant amount of rebroadcasts in comparison to the 1-hop algorithm. In Figures 4.20, 4.21 and 4.22, one may see that the algorithms 1-hop and 2-hop both have lower average latencies than simple flooding.

'circle' topology

Figure 4.15 shows, that in the special case of the 'circle topology' none of the implemented algorithms can save any rebroadcasts. The distance is too large for our algorithms to propagate the needed information about neighborhoods. Figures 4.9, 4.10 and 4.23 to 4.25 confirm our assumption, that all the algorithms perform equally well in this topology.

'line' topology

Figure 4.11 and Figure 4.12 show the reachability distribution and the mean reachability, respectively, of the measurements in the 'line' topology. In this topology the two algorithms 1-hop and 2-hop can save some rebroadcasts only at each the end of the line, they perform only slightly better than simple flooding. The saved rebroadcasts can be seen in Figure 4.16. Apart from some noise not much differences can be seen in the average latencies in the Figures 4.26 to 4.28.

Summary

Both the 1-hop and the 2-hop algorithm perform better than simple flooding in most topologies, as the average reachability is higher for all data rates. Depending on the topology the 2-hop algorithm performs either about equally well or better than the 1-hop algorithm. Average latency measurements contain much noise, especially at high data rates but a performance improvement is still visible in some cases. One has to keep in mind that these measurements were all done in virtual topologies and measurements in real topologies will most likely perform better because the “hidden traffic” from invisible nodes won't be present.

Chapter 5

Future Work

The application developed in this thesis, was written with extendability in mind. It should be easy to add new algorithms to the implementation. Even the networking part can be extended. For instance the Address class could be extended to support IPv6 networking. Thanks to the doxygen documentation, it will be easy for a new developer to start a new project based on this thesis.
The network and the timer classes have been kept strictly separated from the data dissemination part of the code and can easily be used by other projects.

5.1 Sophisticated Simulator

The simple simulator that was implemented in this thesis has much room for improvement. For now, it was only used to validate the code and measurements were performed in a real environment. As described earlier, the simulator does not simulate packet loss due to congestion and thus resulting collisions, it simply drops packets according to a node-to-node configurable probability.
For the development of future data dissemination algorithms, it would be much easier if a simulator which could simulate packet collisions was available. Such a simulator was started, but development was discontinued because finishing it would have consumed too much time. The problems encountered during development were caused mostly by timing issues. That is, the simulator was implemented as a simple networking application, which was listening on a UDP port and the test nodes were sending their packets to that port. That made timing very inaccurate. A lot of packets appear to be sent at the same time and thus the simulation would have resulted in a lot of collisions. This may be due to buffering in the operating system.
It must be evaluated if this problem can be circumvented by using threads to receive these UDP packets, wheter some configuration changes in the operating system can help or if a totally new approach is needed.

5.2 Consider Mobility

In this thesis, mobility has not been taken into account. Future work could consider mobility of the nodes. In combination with sharing GPS data better results may be achieved.

Chapter 6

Summary

Two lightweight data dissemination algorithms have been implemented and got compared against simple flooding. A simple simulator has been implemented for easier code testing and debugging on a single computer. Measurements were performed in real environments. The implemented algorithms could be validated using the simple simulator and the measurement results.

Appendix A

Code Documentation

A.1 Collaboration Graph of the classes

See Figure A.1.

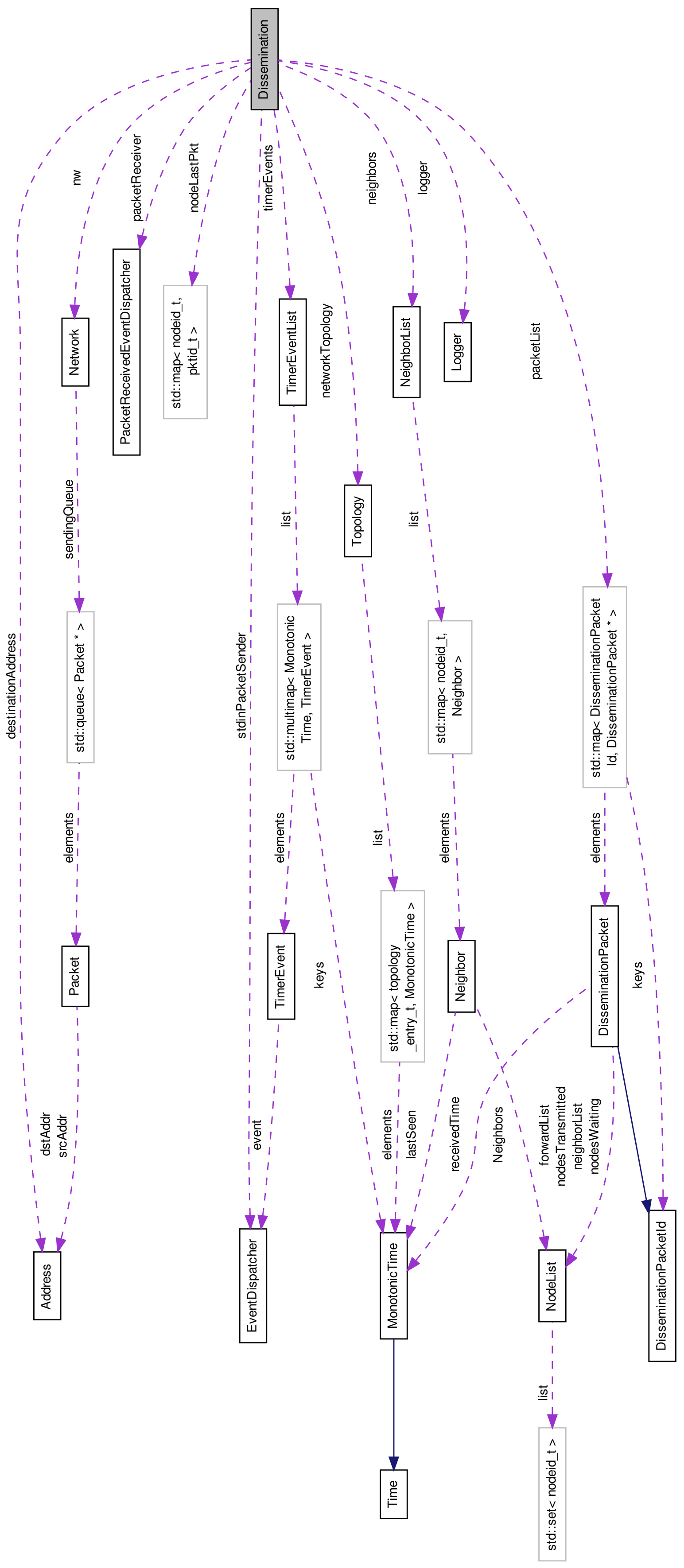

Figure A.1: Collaboration graph

Bibliography

[1] The broadcast storm problem in a mobile ad hoc network,
S. Ni, et al., MobiComm 1999

[2] Spray and wait: An efficient routing scheme for intermittently connected mobile networks,
T. Spyropoulos et al., WDTN '05

[3] Leveraging 1-hop Neighborhood Knowledge for Efficient Flooding in Wireless Ad Hoc Networks,
Y. Cai et al., IPCCC 2005

[4] Location-Aided Flooding: An Energy-Efficient Data Dissemination Protocol for Wireless Sensor Networks,
H. Sabbineni et al., IEEE Transactions On Computers, January 2005

[5] A Geo-location Based Opportunistic Data Dissemination Approach for MANETs,
H. Meyer, K.A. Hummel, Challenged Networks 2009