Anonymous Communications in Mobile Ad Hoc Networks

Huseyin Can

Kongens Lyngby 2006
IMM-M.Sc-2006-91

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-M.Sc: ISSN 0000000000, ISBN 00000000000

Abstract

Based on present technology, it is a challenging task to provide anonymous communications in mobile ad hoc networks. There are several problems that must be addressed properly. Security and performance concerns are the main challenges. Chaum's Mix method can effectively prevent an adversary's attempt of tracing packet routes and hide the source and/or destination of packets. However, applying the Mix method in ad hoc networks may cause significant performance degradation due to its non-adaptive Mix route selection algorithm. The goal of this project is to develop a Mix route algorithm to find topology-dependent Mix routes for anonymous connections. We have named the protocol that will be implemented *MixRoute*. The protocol is implemented in ns-2 [27]. Test scripts are developed to make a simulation.

Preface

This thesis was prepared at Informatics and Mathematical Modelling, the Technical University of Denmark in fulfillment of the requirements for acquiring the M.Sc. degree in engineering.

The thesis deals with designing an anonymous communications protocol suitable for Mobile Ad Hoc Networks. The protocol is later implemented, using C++ and OTcl, into the network simulator software ns-2.

The thesis consists of a report, source code for the protocol and scripts used to test the protocol in ns-2.

Lyngby, September 2006

Huseyin Can

Acknowledgements

I am very grateful for the advice and support from my supervisor, associate professor Christian Damsgaard Jensen, for his feedback and for keeping me focussed in my research. His comments to the report in the final stages have been invaluable.
I would also like to thank my wife for her support and patience during the years and hopefully for many years to come.
Last but not least I want to thank my parents for helping me always keeping focused on school and for the supports when needed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A wireless mobile ad hoc network is formed by a group of mobile hosts that communicate through radio transmissions, without support of fixed routing infrastructure. Due to its ease of deployment, it has a large amount of applications in military as well as in civilian environments. However, wireless medium introduces great opportunities for eavesdropping of wireless data communications. Anyone with the appropriate wireless receiver can eavesdrop and this kind of eavesdropping is virtually undetectable. So communication privacy is one of the issues that a network designer must address with higher priority.
By definition, privacy means the protection of data from unauthorized parties. Federrath et al. [10] discuss the communication privacy requirements in mobile networks in terms of content, location and identity privacy. Content privacy, i.e. protection of the contents of a message, can be provided by encryption schemes (such as AES, DES and RSA). In the wireless ad hoc network following is considered: node address itself does not contain location information, but may disclose identity of mobile users. An adversary may learn user communication patterns such as who communicates with whom, when, how long, etc. from traffic information. To prevent traffic analysis, it is desirable that user communications remain anonymous. How to provide anonymity support in wireless ad hoc network is the topic of this project.
Achieving anonymity is a different problem than achieving data confidentiality. While data can be protected by cryptographic means, the recipient node address (and maybe the sender node address) of a packet can not be simply encrypted

because they are needed by the network to route the packet. Most existing anonymizing schemes are originated from Chaum's Mix-net concept [6]. The idea is that traffic sent from sender to destination should pass one or more Mix nodes. A Mix node relays data from different end-to-end connections, and its task is to reorder and re-encrypt the data such that incoming and outgoing data cannot be related. This should prevent attempts of an outside eavesdropper to follow an end-to-end connection. A Mix-net can protect against colluding Mix nodes if not all Mix nodes involved in relaying an end-to-end connection collude with the adversary. This is an important property because, in a hostile environment (e.g., battlefield), the probability that roaming nodes be captured cannot be neglected. Generally, the more Mixes are involved in relaying an end-to-end connection, the lower the probability that the connection be compromised. However, relaying data traffic through too many Mixes would inevitably increase the average data latency and decrease the average data delivery ratio. So the number and sequence of Mixes in the path of an end-to-end connection must be appropriately determined in order to reach a balance between the two contradictory goals. This is the so-called Mix routing problem.
Mix routing has not received sufficient attention in the design of existing Mix-based anonymizing systems. The reason behind this is that most existing systems are designed for operating over the Internet. One class of anonymizing systems is represented by Onion Routing [30], where the Mix set is small, all Mixes are administered by a central authority, and the Mix-net topology remains stable during run time [27, 4]. Another class of anonymizing systems that emerged recently is of peer-to-peer type [31, 11], where all participating nodes are potential originators of traffic as well as potential relays. Since a peer-to-peer anonymizing network has a very large node base, an adversary cannot observe the entire network. A Mix route can be constructed as follows. The source node of an end-to-end connection chooses the first Mix from its neighbor set, which then chooses the second Mix similarly, and so on. The Mix route length can be controlled by the source node [11], or by the last Mix based on a probability of forwarding [31]. The biggest challenge of Mix routing in wireless ad hoc network is the dynamic change of topology, which makes a static or random Mix route inefficient.

1.1 Goals of the project

The goal in this project is to make improvements in Mix-net performance by proposing a Mix route algorithm which adapts to topology change. To do this we have to find the state of art in anonymous communications in mobile ad hoc networks. Based on the state of art we will design an improved Mix route algorithm which will be an enhancement to Chaum's Mix method. After designing

the algorithm, it will be implemented into a network protocol in ns-2. We will also make a simulation model that can be used to test the implementation of the network protocol.
The report is organized as follows. First ad hoc networking is introduced in chapter 2 where we will look closer to the definition of ad hoc networks. There will be presented example scenarios where an infrastructure is not available, and where ad hoc networks are suitable. Thereafter known routing protocols for ad hoc networks will be described. Finally, a summary will be given where the most suitable choice of routing protocol for our project is presented.
State of art in anonymous communications in general is given in chapter 3. In this chapter all recent anonymity designs are presented. They are divided into three categories:

1. Proxy Services
2. Chaumian Mix-nets
3. Remailers: SMTP Mix-nets

Then there will be given a presentation of recent designs where Mix-nets are used in wired networks. Lastly, a summary of the chapter is given.
In chapter 4 state of art in anonymous communications in ad hoc networks is given. A summary will provide the pros and cons of the state of art in MANETS.
In chapter 5 the privacy requirements of the project is presented. A summary will address the requirements for the design of MixRoute.
In chapter 6 the design of our protocol is provided, designed for ad hoc networks, and then a qualitative cost and a security analysis is conducted.
We will present a short presentation of ns-2 in chapter 7.
After the presentation of ns-2, in chapter 8 the implementation of the protocol is described in details.
After the implementation of the protocol a simulation model is designed to test the protocol, which is provided chapter 9.
The evaluation of the protocol is described in chapter 10. In this chapter the test scenario will be presented, and expectations of test results will also be provided.
Finally a conclusion is given in chapter 11. In this chapter a conclusion of the report is given.
The source code of the protocol and test scripts are provided in the appendix.

Chapter 2

Ad Hoc Networks

In this chapter the concept of ad hoc networks is introduced. The questions of when this could be useful are also addressed even though it will not be completely drained. There is a large potential in future applications to use it. Furthermore the challenges in ad hoc networking are up for discussion together with a short description of the commonly used algorithm for routing.

2.1 Definition of Ad Hoc Networks

Users of networked technology, which is an ever growing number of people and machines, are getting more and more accustomed to constantly being able to access different on-line services. It may be e-mail or on-line dictionary services, ticket booking, or traveling information such as road maps and driving directions.
The networks for mobile phones are usually available in most inhabited areas. As new technologies are developed even more services are available through these networks. Even high capacity hot spots are being common practise in densely populated cities, mostly at hotels and airports even though some gas stations are getting hot-spots installed. This will of course continue to expand to more areas.
Techniques used for wireless connection is still dependent on base stations to

connect to. These base stations are in turn connected to an infrastructure. To be able to expand the wireless services they depend on this infrastructure. Problems arise when such infrastructure is not available. It can be costly to build new links for which there is little profit for the service provider.
What we would like is the ability to connect to available services without the need for an infrastructure. These spontaneous connections are, as the name implies, the ad hoc part of the networking.
Solutions exist for mobile users to connect to each other through the Internet. This can be accomplished using DHCP[1] and Mobile IP. This does, however, depend on the availability of servers that allow the users computers to connect. Using Mobile IP (MIP) the information between two users in the same room even gets routed and tunnelled through the Internet.
Throughout the history of ad hoc networks they have also been called network on-demand or mesh networks. Although given these names they are of similar operational ideas. The ad hoc networks have been on the research desk for a long time but have recently gained more interest. The military applications of such networks have seeded new interest into the research community. Initially this type of networking was researched by the military.

2.2 Motivation of Ad Hoc Networks

There exists numerous occasions where an infrastructure is not available. This section will present some of the example scenarios where ad hoc networks might come in handy. The basic ideas are possible commercial usages [24].

2.2.1 Emergency Services

Anywhere when there is an emergency there is a need to co-ordinate the rescue personnel. This is commonly solved using hand held or vehicle mounted radios. However, what about the infrastructure that may have been damaged and is no longer in operation? To quickly get things going again the use of ad hoc networks can automatically fix this.
This might not be such a big problem in small fires or so, but when larger areas are hit by a natural disaster it can be important to quickly be able to communicate. By using ad hoc networks to set up a network infrastructure it is simply a matter of placing out a couple of mobile routers which makes it easy and fast.

[1] Dynamic Host Configuration Protocol, RFC 2131, March 1997

2.2.2 Conferences

In many situations the need for connecting and exchanging information between participants of a conference or some other meeting is clear. Usually there is a great need for collaboration and since the home network environment is not available there is a need for other solutions.
There are usually available networks for the participants to use but this might imply very large round trips for the data using for example Mobile IP [24].

2.2.3 Home Networking

Given that the use of wireless computers and appliances keeps on growing in the home environment the need for helping out administrating this is also expanding. Using the techniques of ad hoc networks that configures themselves is truly something that would be of great help.
Also, if the computers are used at more places than at home, at the office or school maybe, there is still larger administrative burden that must be kept down.

2.2.4 Personal Area Networks

Many objects that are tightly coupled to a single person can take advantage of being connected to each other forming a personal area network. The network itself is most definitely mobile since people tend not to stay around for long in one spot. This makes the use of ad hoc in personal area networks less needed. However, when getting connected to another personal area network (PAN) the connections between persons devices might be wanted. In this case there is definitely a need for ad hoc networking support.

2.2.5 Embedded Systems

As more and more machines everywhere is in need for communicating different things to the surroundings a need for ad hoc networking arises. One can think of objects that can respond to changes in the environment and together with other devices perform different scenarios depending on the current context.
It might be a toy with built in networking capabilities that can interact with the home computer to lookup some data at the Internet or a connected phone that can turn down the volume of the stereo and TV when there is an incoming call.

Some researchers are thinking about ubiquitous computing, where we will have computers connected to each other performing tasks depending on changing environment all around us. It is hard to see every possible use of this technology at the current time, but new services and applications will surely benefit of having ad hoc network support.

2.2.6 Sensors

Using tiny devices that are able to gather different information such as temperature, concentrations of different chemicals and gasses, vibrations, and so on can be of importance in accidents and emergency situations. Constructing these sensors so that when turned on they form an ad hoc network and report back to a well known data collecting node they can be of great importance.
For example in the case of a gas leak, instead of sending rescue personnel into the dangerous area these sensors can be dropped from an air plane or helicopter. The use of the gathered data can be helpful in devising a plan to take care of the situation.
In the military field there can of course be a lot of applications for these kinds of data collecting devices.

2.3 Routing

The routing of data packets in computer networks is the process of moving information from a source to a given destination. The way to do this in the best way possible is the concern of the routing algorithm used. In the case of wired networks the main routing algorithms used are either distance vector routing or link state routing.

2.3.1 General Routing Principles

Routing has been going on in large networks, as the Internet, for quite some time and lots of different routing algorithms has been proposed and tested. The main categories that have survived and have been in large use are those described below. These are performing well in wired networks, but some problems still exist.

2.3.1.1 Distance Vector Routing

A distance vector routing algorithm is a distributed algorithm that is quite simple to implement and has low computational demands. It basically means that each router has a way of knowing about its neighbors. The router knows the metric of each link to those neighbors. The metric can be of any type, for example delay or similar.
Also, each router has a distance vector, that is, a table of distances, to each destination available and which outgoing link to use. All the neighboring routers exchange information between each other. Based on what they know about their neighboring router it updates its own vector with the minimum distances. If some values are changed the router in turn, sends out an update to all its neighbors.
Distance vector routing is commonly known to have problems with changing topology. Especially a problem known as the count-to-infinity problem which makes the routers construct an ever growing path to a node that has gone down or a link to that node being broken. The algorithm can also converge slowly on changing topology, and while converging, create routing loops [35].

2.3.1.2 Link State Routing

Instead of depending on every neighbor to gradually give information about how to get all the destinations the link state routing algorithm gets a complete picture of the entire network and locally calculates the shortest path to every destination. This means that for each router to get information about every link all the routers must broadcast (flood) the information that they have to all others.
When all the information needed is collected the process of finding the shortest path can begin. This is accomplished using Dijkstra's or Prim's algorithm. To be able to compute the shortest path all nodes need to have a complete view of the network and then perform the computations locally. This makes these kinds of algorithms global routing algorithms in comparison to the distributed or decentralized distance vector algorithm.
As has been stated this kind of algorithm demands a complete view over the network which makes the number of messages needed to be sent between all routers relatively high. Also, the computations of the shortest path problems using Dijkstra's algorithm is in the order of $O(n^2)$ [19].

2.3.1.3 Comparison

As was described above the distance vector routing algorithms are distributed, relatively easy to implement, and does not need much processing power or memory. In the contrary, link state routing demands more bandwidth to distribute all information between all routers, more memory to hold the complete topological graph, and more processing power to compute the final result.
However, the distance vector approach does have some problems with slow convergence on changing topologies. There are of course some problems with the link state variant also, for example how to know when all information has been collected and whether all nodes are working with the same overview or are doing their calculations on different topologies.
What is gained with the link state approach is robustness. If one router gets the wrong idea of the network topology or calculates the wrong routes it can damage some paths. But if a router using the distance vector algorithm gets the wrong idea about some link status it is spread to all the other routers and they have no idea what is right or wrong [19].

2.3.1.4 Scaling and the Hierarchical Approach

There is a problem with both mentioned algorithms, they scale relative poorly. The algorithms described above need a distance to all possible destinations. On large networks this is a big problem. But there are solutions.
By dividing the network into different areas and layers it is possible to route the messages within each area separately. If a message needs to go to another area it is forwarded to an area gateway which in turn routes messages between the other area gateways and so on. Using this technique it is possible to cut down the number of destinations each sub level router needs to know about thus saving memory, bandwidth, and processing power.

2.3.2 Routing in Ad Hoc Networks

In the case of a mobile ad hoc network the topology is highly dynamic. This leads to quickly changing link states. Some links get broken while other links are created by other pairs of routers as is depicted in Figure 2.1. In this picture the mobile host 1 (MH_1) is moving from the vicinity of MH_2. As it gets closer to MH_7 and MH_8 new links are established to these hosts. These characteristics are different from the one that appears in most wired networks. The routing algorithms used in the wired case have problems with topology changes, and if these happen often the problems are just getting worse.

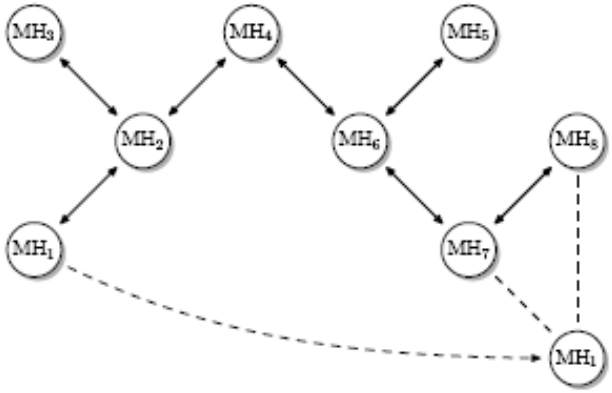

Figure 2.1: An ad hoc network of mobile nodes

Another problem that arises in wireless networks that is not as common in wired routing is the asymmetrical links. That is, one node can reach another but the return path is not the same. Some ad hoc routing algorithms described below handles this and some do not.
The following sections describe different solutions and main ideas about routing in ad hoc networks that overcome, or at least damps, some of the routing problems. This is a short introduction that emphasizes the different needs.

2.3.2.1 Destination-Sequenced Distance Vector Protocol

Destination-Sequenced Distance Vector (DSDV) is an ad hoc version of the commonly known distance vector algorithm. To overcome the problems with slow convergence in the ordinary distance vector algorithm and prevent routing loops in highly dynamic topologies this algorithm adds a sequence number to the routing table entries.
The sequence numbers are updated by the destination nodes when new links to them are detected. Also, when a node detects a broken link it sends this out together with an updated sequence number. The receiving nodes check for higher or equal sequence numbers. If a route update packet is received with lower sequence number it is discarded. In the case of a sequence number that is equal to the one already held the metric is checked to see if it is better or worse. To save bandwidth the protocol uses two types of route update messages. First one is a full dump that a node sends to all its neighbors. These messages contain the complete routing table. The other variant is an incremental update which only updates the routes that has changed since the last full update. In this way

it is possible to send only small packets, conserving bandwidth and transmission time for most cases. When these incremental updates are getting too big the node can send a full dump instead in hope to be able to send smaller packets after that.
To get rid of the possibility of an oscillating system update messages are only sent out after a delay. After this delay the routing information has stabilized and is not that sensitive to oscillation. The delay is computed using a running, weighted average over the most recent updates.
Since the topology might change the route update messages are sent out at certain time intervals. However, there is no need for synchronizing the different nodes as the update events are handled asynchronous. The use of these repeating update messages keeps all the nodes busy when not in need for communication. However, when the needs arrive all nodes are ready to forward the data directly. The DSDV algorithm gets rid of the undesirable properties that the original algorithm possesses. It propagates the bad news of broken links fast and keeps the path updates stable. Also, using the sequence number rules for updating distance vector values it guarantees loop-free paths to each destinations at all times [23].

2.3.2.2 Dynamic Source Routing

The Dynamic Source Routing (DSR) protocol is completely demand based. It does not need any kind of periodic updates or node announcement messages. Instead, the protocol acquires the needed routing information on-demand.
The routing protocol is divided into two parts. The first is the route discovery and the second is route maintenance. The discovery phase is initiated when a node needs to send information to another node that is not available in its current path cache. The node broadcasts a special discovery packet with the destination and a unique identification number. The packet is received by all nodes within the wireless transmission range. They, if they are not the destination, add their node address to the path in the packet header and retransmit the discovery packet. A packet with the same identity as has already been seen is discarded. Also, if the node itself is mentioned in the path header the packet is discarded. This technique efficiently cuts down on duplicate packets in the air.
When the packet finally finds its way to the destination, the destination node returns a route reply. The reply is sent using the routing cache if present. Otherwise a new route discovery is initiated but in this case with the route reply piggy backed on the discovery packet. If this piggybacking is not allowed there is a great risk of infinite looping of route discoveries. Another way is to reverse the source path collected by the route discovery; however, this takes for granted that all links are bidirectional which may not always be the case. A simple ex-

ample of the broadcast of source route discovery packets is shown in Figure 2.2. After a path has been discovered the second phase of the protocol is in use. This is the maintenance phase. During this phase all communications are done using the previously found paths. Each node on the path is responsible for resending packets that are not acknowledged by the next hop node. After a maximum limit number of retries a route error message is sent back to the source node indicating that the path is broken.

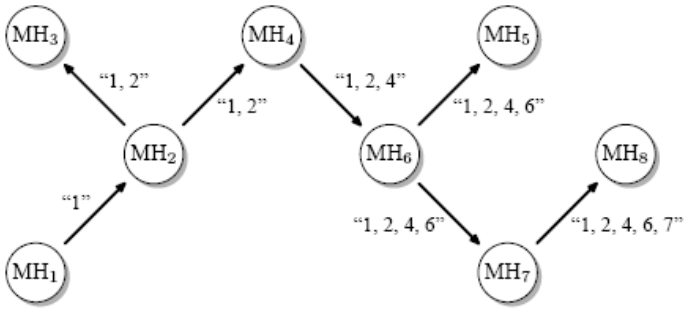

Figure 2.2: DSR route discovery example

A number of different optimizations of the protocol are possible. For example, each node on a path that is not actually originating a route discovery can cache some part of the messages that they see go by. This can speed up route discovery but may also place some overhead on the software and the processing time of the processor [16].

2.3.2.3 Ad Hoc On-Demand Distance Vector Routing

In contrast to the DSDV, the Ad hoc On demand Distance Vector (AODV) is an on-demand protocol. It borrows the idea of route discovery and maintenance while still using a distance vector approach to the routing.
Each node keeps a distance vector for each known destination and its next hop neighbor. If a destination that is needed is not in the list a route discovery is initiated. By issuing a broadcast packet to its neighbor containing source and destination addresses and a sequence number the node hopes to find a path to the destination.
If a node, that is not the actual destination, receives a request it checks its own routing table. If it can find the destination there it replies to the source node with this information. If, however, this information is not in the node's routing table it adds the source as a destination in its own table using appropriate hop count, increments the request packet's hop count and rebroadcasts it to its

neighbors.
This procedure is continued until the destination is reached and a route reply packet is sent back to the source. After this all data traffic is routed using the discovered path. If the nodes move and some links break a route error packet is sent out to tell the nodes that the path is no longer available. If this happens the source node initiates a new route request. An example of a route discovery and path setup is shown in Figure 2.3.

	Route Request			**Route Reply**	
1	$S \to *:$	$\langle \text{RREQ}, S, D, 0, S \rangle$		$D \to C:$	$\langle \underline{\text{RREP}}, S, D, \underline{0}, \underline{D} \rangle$
2	$A:$	$d[S] = 1, n[S] = S$	6	$C:$	$d[D] = 1, n[D] = D$
	$A \to *:$	$\langle \text{RREQ}, S, D, \underline{1}, \underline{A} \rangle$		$C \to B:$	$\langle \text{RREP}, S, D, \underline{1}, \underline{C} \rangle$
3	$B:$	$d[S] = 2, n[S] = A$	7	$B:$	$d[D] = 2, n[D] = C$
	$B \to *:$	$\langle \text{RREQ}, S, D, \underline{2}, \underline{B} \rangle$		$B \to A:$	$\langle \text{RREP}, S, D, \underline{2}, \underline{B} \rangle$
4	$C:$	$d[S] = 3, n[S] = B$	8	$A:$	$d[D] = 3, n[D] = B$
	$C \to *:$	$\langle \text{RREQ}, S, D, \underline{3}, \underline{C} \rangle$		$A \to S:$	$\langle \text{RREP}, S, D, \underline{3}, \underline{A} \rangle$
5	$D:$	$d[S] = 4, n[S] = C$	9	$S:$	$d[D] = 4, n[D] = A$

Figure 2.3: Simplified route discovery example in AODV.

To eliminate the need of flooding a large network with route request packages the time to live field is used to limit the number of hops a route request can be sent. If a request fails this is gradually incremented until the destination is reached. This approach is called expanding ring search.
The AODV protocol can only handle symmetrical links but has the ability to also be used in multi cast groups. Additionally it supports a hello message, a variant of the route request message, which is used to detect the one hop neighbors. This may not be used if the lower layers already support this kind of service [22].

2.3.2.4 Cluster-Based Networks

In some situations the network may be small, that is, consist of a small number of nodes. In this case the above mentioned routing algorithms may be adequate. However, if the network grows and the number of nodes are large there may be unnecessary overhead and the network diameter, the number of hops, may become too large to handle straight on.
In the case of wired networks area controllers are used to manage a limited set of computers and routing between these are done using some backbone infrastructure. This makes the network manageable for each smaller cluster and the

network diameter might become smaller using the backbone routers. However, the setup of these area controllers is done by administrative personnel often during the design of the network. This is clearly not something that can be done as easily in mobile ad hoc networks. There are some ideas and techniques available that manages to accomplish these goals automatically using distributed algorithms and elections of cluster heads.
The idea of using clusters is beneficial in many ways. It can be used to group nodes into clusters that are separated to nearby clusters by transmission frequency or spread spectrum coding. This way the congestion can be lowered [32].

2.3.2.5 Zone Routing Protocol

A common approach in many fields is to take the best ideas and combine them. This is one such idea. The zone routing protocol (ZRP) uses a proactive routing variant within a limited zone defined by a small number of hops around each node. Outside this zone a reactive protocol is used.
Proactive means that all the routes are known by all the nodes before any data is needed to be sent. The link state and distance vector protocols are based on these assumptions. This is useful in the ZRP because of the limited number of nodes in each zone. These zones are locally defined by every node and thus highly overlapping. Also, the idea is that there is more often communications taking place within a relatively small geographical area than to distant nodes. This may be true in some setups, for example military tactical units of emergency rescue teams, while other applications might have different demands.
The mobility of nodes affects are only local since the routing between nodes are done reactively or on-demand, like AODV or DSR. The routing to a distant zone, therefore, is not affected by local link movements as much as for other protocols.
The implementation of the two different routing protocols needed can be any pair of protocols. The two variants are called Interzone Routing Protocol for the routing between zones and Intrazone Routing Protocol for the routing within a local zone [2].

2.4 Challenges in Ad Hoc Networking

As we have seen in the previous sections there are many different approaches to the routing dilemma in fast changing topology networks as mobile ad hoc networks. Even though there are a lot of different approaches to consider they fit well into different needs.

One thing that has not yet been surfaced is the need for addressing. The routing protocols are working with unique node addresses, for example IP number. These addresses must, however, be handed out in some way. Also, the need for gateways to wired networks needs to be considered in the addressing schema.
A big challenge is of course to keep the routing tables needed up to date with a fast changing topology. Also, the problem of loop freedom and scarce bandwidth available puts even higher demands on the routing algorithm. On top of this the size of the networks can be from just a few nodes to over hundreds of them making the routing algorithms sensible for some scaling problems.
On the commercial market the use of ad hoc networking techniques have only started to be available in some networks. Most notably within the companions own wireless networks or building to the building links. The use in more everyday products such as mobile phones have low commercial interest since the operators of the networks probably will lose some of the traffic. Also, not every one is happy with having their mobile phone forwarding traffic for someone else. Not because of the traffic itself but because the battery power drain. The largest possibility for the commercial breakthrough is probably within the wireless local area networks where the standards already has support for some limited ad hoc connections, or peer to peer support.
The use of multi hop wireless networks can help keeping the power consumption down due to lowering the link length. However, the need to make the routing protocols power aware and not waste too much power on control messages instead of actual information traffic is essential. Also, multi hop networks are dependent on the intermediate nodes being available even though that node may not be in transceiver mode. The use of multiple available routes might be a solution where some nodes can go down in power saving mode while others peek up now and then to sense the communications.

2.5 Summary

In this thesis the focus will be on anonymity in mobile ad hoc networks based on an underlying routing protocol. So we must not forget that the routing protocol has to be suitable for anonymous communication. But in the case of choosing the routing protocol our main goal is to use the most suitable for MANETS.
We have now studied the known routing protocols suitable for MANETS. The first thing that has to be decided is if the protocol should be reactive or proactive.

2.5.1 Proactive protocols

In the proactive protocols routes are set up based on continuous control traffic, and all routes are maintained all the time. Here is listed the pros and cons of proactive protocols:

- Constant overhead created by control traffic
- Too demanding on the power consumption
- Routes are always available

It is a high price to pay for constant availability of routes, when the power consumption is high especially in large networks. In a MANET nodes use batteries and it can be a demanding task if batteries have to be charged too often. There is also another problem with an proactive protocol; when there is constant route maintenance it is also easier to compromise the network and uncover the identity of nodes. In fact the anonymity will be compromised. For these reasons we chose not to use a proactive protocol such as DSDV.

2.5.2 Reactive protocols

The reactive protocols do not take initiative for finding routes, and establish routes on-demand by flooding a query. The pros and cons for reactive protocols are listed below:

- Does not use bandwidth except when needed (when finding a route)
- Much network overhead in the flooding process when querying for routes
- Initial delay in traffic

For large networks it is a better approach to use a reactive protocol. The power consumption is minimal and the energy level will not be drained as quickly as in a proactive protocol. In the initial process when the network is flooded it is easier to make traffic analysis, but this problem is going to be solved by using Mix nodes. Based on these studies we chose to use a reactive protocol to keep the power consumption minimal. In a network where the nodes move frequently it is also optimal to use a reactive protocol, because we only flood the network once (in the initial process) and therefore we don't need to keep tracking the

new positions of the nodes. For this reason we chose to use a reactive protocol as our underlying routing protocol. Both AODV and DSR are good solutions. The main difference between AODV and DSR is that in DSR a source routing option is used; i.e. when a node wants to send something to a destination it sets the whole route for that packet, indicating the addresses of the nodes it has to pass through. In this sense all packets have a DSR header included, and it is needed that all nodes within the ad hoc network know the whole network topology. On the other hand, AODV does not perform source routing at all; when a node wants to send something to a destination, it checks its routing table, looking for the next hop towards that destination, and sends the packet to it, and so on. In this sense, data packets "travel" through the ad hoc network without any AODV specific information. The problems with the AODV is that it uses more, but smaller routing control packets → critical concerning wireless medium properties (e.g. interference). This becomes worse for a higher load, as neighbors have to be rediscovered (congestion causes link failures).
DSR has some problems concerning the cache usage: The advantage of multiple routes becomes a disadvantage with high mobility. In bigger networks, the source-routing principle can also become a problem.
Based on the knowledge of both AODV and DSR we chose to use DSR as the underlying routing protocol because of the lesser use of control packets. The compromise we have to make for using DSR is acceptable in our case.
In the next chapter the state of art in recent anonymity designs will be described, and there will be given concrete examples.

Chapter 3

Recent Anonymity Designs

For anonymous routing we have to find an optimal routing technique. In chapter 2 we investigated the existing routing methods in MANET's which could be applied to our design. To gain knowledge about the state of art in the area of anonymous communications in general terms several areas have to be investigated.
We review three main types of design: proxy-servers, mix-nets, and other anonymous communications channels.

3.1 Proxy Services

Proxy services provide one of the most basic forms of anonymity, inserting a third party between the sender and recipient of a given message. Proxy services are characterized as having only one centralized layer of separation between message sender and recipient. The proxy serves as a "trusted third party", responsible for sufficiently stripping headers and other distinguishing information from sender requests.
Proxies only provide unlinkability between sender and receiver, given that the proxy itself remains uncompromised. This unlinkability does not have the quality of perfect forward anonymity, as proxy users often connect from the

same IP address. Therefore, any future information used to gain linkability between sender and receiver (i.e., intersection attacks, traffic analysis) can be used against previously recorded communications.
Sender and receiver anonymity is lost to an adversary that may monitor incoming traffic to the proxy. While the actual contents of the message might still be computationally secure via encryption, the adversary can correlate the message to a sender/receiver agent.
This loss of sender/receiver anonymity plagues all systems which include external clients which interact through a separate communications channel - that is, we can define some distinct edge of the channel. If an adversary can monitor this edge link or the first-hop node within the channel, this observer gains agent-message correlation. Obviously, the ability to monitor this link or node depends on the adversary's resources and the number of links and nodes which exist. In a proxy system, this number is small. In a globally-distributed mixnet, this number could be very large. The adversary's ability also depends on her focus: whether she is observing messages and agents at random, or if she is monitored specific senders/receivers on purpose.

3.1.1 Anonymizer.com

The Anonymizer was one of the first examples of a form-based web proxy. Users point their browsers at the Anonymizer page at www.anonymizer.com. Once there, they enter their destination URL into a form displayed on that page. The Anonymizer then acts as an http proxy for these users, stripping off all identifying information from http requests and forwarding them on to the destination URL.
The functionality is limited. Only http requests are proxied, and the Anonymizer does not handle cgi scripts. In addition, unless the user chains several proxies together, he or she may be vulnerable to an adversary which tries to correlate incoming and outgoing http requests. Only the data stream is anonymized, not the connection itself. Therefore, the proxy does not prevent traffic analysis attacks like tracking data as it moves through the network [5].

3.1.2 Lucent's Proxymate

Chaining multiple proxies together by hand is a tedious business, requiring many preliminaries before the first web page is reached. Lucent's Proxymate software automates the process. The software looks like a proxy sitting on the user's computer. By setting software to use the Proxymate proxy, the user causes the software's requests and traffic to go to the software, which then automatically

negotiates a chain of proxies for each connection [12].

3.1.3 Proxomitron

Another piece of software which helps manage many distinct proxies in a transparent manner is Proxomitron. In addition to basic listing and chaining of proxies, Proxomitron allows users to write filter scripts. These filters can then be applied to incoming and outgoing traffic to do everything from detecting a request for the user's e-mail address by a web site to automatically changing colors on incoming web pages [28].

3.2 Chaumian Mix-nets

The project of anonymity on the Internet was kicked off by David Chaum in 1981 with a paper in Communications of the ACM describing a system called a "Mix-net". This system uses a very simple technique to provide anonymity: a sender and receiver are linked by a chain of servers called Mixes. Each Mix in the chain strips off the identifying marks on incoming messages and then sends the message to the next Mix, based on routing instructions which encrypted with its public key. Comparatively simple to understand and implement, this Mix-net (or "mix-net" or "mixnet") design is used in almost all of today's practical anonymous channels.

3.2.1 Chaum's Digital Mix

Chaum presents a solution to the traffic analysis problem that is based on public key infrastructure. A message X is sealed with a public key K so that only the holder of the private key K^{-1} can discover its content. If X is simply encrypted with K, then anyone could verify a guess that $Y = X$ by checking whether *K(Y) = K(X)*. This threat can be eliminated by attaching a large string of random bits R to X before encrypting. The sealing of X with K is then denoted *K(R,X)*. The users of the cryptosystem will include not only the correspondents but a computer called a *mix* that will process each item of mail before it is delivered. A participant prepares a message M for delivery to a participant at address A by sealing it with the addressee's public key K_a, appending the address A, and then sealing the result with the mix's public key K_1. The *mix* decrypts its input with its private key, throws away the random string R_1m and outputs the

remainder. One might imagine a mechanism that forwards the sealed messages $K_a(R_0, M)$ of the output to the addressees who then decrypt them with their own private keys. The purpose of a mix is to hide the correspondences between the items in its input and those in its output. The order of arrival is hidden by outputting the uniformly sized items in lexicographically ordered batches.
Chaum also defines how senders apply a return address in the message field, which only the destination can determine with its private key. The process of sending a message from destination to source is the same as mentioned above [6].

3.2.2 ISDN Mixes

Chaum's original Digital Mix was described in terms of a series of Mix nodes which passed idealized messages over a network. The first proposal for the practical application of mixes came from Pfitzmann et. al., who showed how a mix-net could be used with ISDN lines to anonymize a telephone user's real location. Their motivation was to protect the privacy of the user in the face of a telephone network owned by a state telephone monopoly.
Their paper introduced a distinction between explicit and implicit addresses. An explicit address is something about a message which clearly and unambiguously links it to a recipient and can be read by everyone, such as a To: header. An implicit address is an attribute of a message which links it to a recipient and can only be determined by that recipient. For example, being encrypted with the recipient's public key in a recipient-hiding public key is an implicit address [25].

3.3 Remailers: SMTP Mix-nets

In earlier days the anonymous remailer was a popular anonymous communication form. Remailers are divided into three categories Type 0, Type 1 and Type 2.

3.3.1 Type 0: anon.penet.fi

One of the first and most popular remailers was anon.penet.fi, run by Johan Helsingius. This remailer was very simple to use. A user simply added an extra header to e-mail indicating the final destination, which could be either an e-mail address or a Usenet newsgroup. This e-mail was sent to the anon.penet.fi server, which stripped off the return address and forwarded it along. In addition, the

server provided for return addresses of the form "anXXXX@anon.penet.fi"; mail sent to such an address would automatically be forwarded to another e-mail address. These pseudonyms could be set up with a single e-mail to the remailer; the machine simply sent back a reply with the user's new pseudonym.
The anon.penet.fi remailer is referred to as a Type 0 remailer for two reasons. First, it was the original "anonymous remailer." More people used anon.penet.fi than are known to have used any following type of remailer. Exact statistics are hard to come by, but X number of accounts were registered at penet.fi, and only Y are currently registered at nym.alias.net.
Second, anon.penet.fi did not provide some of the features which motivated the development of "Type I" and "Type II" remailers. In particular, it provided a single point of failure and the remailer administrator had access to each user's "real" e-mail address. In general, any remailer system which consists of a single hop is considered Type 0.
This last feature proved to be the service's undoing. The Church of Scientology, a group founded by the science fiction writer L. Ron Hubbard, sued a penet.fi pseudonym for distributing materials reserved for high initiates to a Usenet newsgroup. Scientology claimed that the material was copyrighted "technology." The poster claimed it was a fraud used to extort money from gullible and desperate fools. Scientology won a court judgment requiring the anon.penet.fi remailer to give up the true name of the pseudonymous poster, which the operator eventually did. This incident, plus several allegations of traffic in child pornography, eventually convinced Johan Helsingius to close the service in 1995.
Services similar to Type 0 remailers now exist in the form of "free e-mail" services such as Hotmail, Hushmail, and ZipLip, which allow anyone to set up an account via a web form. Hushmail and ZipLip even keep e-mail in encrypted form on their server. Unfortunately, these services are not sufficient by themselves, as an eavesdropping adversary can determine which account corresponds to a user simply by watching him or her login.

3.3.2 Type 1: Cypherpunks Remailers

The drawbacks of anon.penet.fi spurred the development of "cypherpunks" or "Type 1" remailers, so named because their design took place on the cypherpunks mailing list. This generation of remailers addressed the two major problems with anon.penet.fi: first, the single point of failure, and second, the vast amount of information about users of the service collected at that point of failure. Several remailers exist; a current list can be found at the Electronic Frontiers Georgia site or on the newsgroup alt.privacy.anon-server.
Each cypherpunk remailer has a public key and uses PGP for encryption. Mail can be sent to each remailer encrypted with its key, preventing an eavesdropper from seeing it in transit. A message sent to a remailer can consist of a request to

remail to another remailer and a message encrypted with the second remailer's public key. In this way a chain of remailers can be built, such that the first remailer in the chain knows the sender, the last remailer knows the recipient, and the middle remailers know neither.
Cypherpunk remailers also allow for reply blocks. These consist of a series of routing instructions for a chain of remailers which define a route through the remailer net to an address. Reply blocks allow users to create and maintain pseudonyms which receive e-mail. By prepending the reply block to a message and sending the two together to the first remailer in the chain, a message can be sent to a party without knowing his or her real e-mail address [8].

3.3.3 Type 2: Cottrell's Mixmaster

This remailer addresses some of the problems with Type 1 remailers:

- Traffic Analysis: Cypherpunk remailers tend to send messages as soon as they arrive, or after some specified amount of delay. The first option makes it easy for an adversary to correlate messages across the mix-net. It's not clear how much delay helps protect against this attack.
- Does Not Hide Length: The length of messages is not hidden by the encryption used by cypherpunk remailers. This allows an adversary to track a message as it passes through the mixnet by looking for messages of approximately the same length.

Instead of using PGP, Mixmaster uses its own client software (which is also the server software), which understands a special Mixmaster packet format. All packets are the same length. Every message is encrypted with a separate 3DES key for each mix node in a chain between the sender and receiver; these 3DES keys are in turn encrypted with the RSA public keys of each mix node. When a message reaches a mix node, it decrypts the header, decrypts the body of the message, and then places the message in a "message pool". Once enough messages have been placed in the pool, the node picks a random message to forward [21].

3.3.4 Nymserver and nym.alias.net

The reply blocks used by cypherpunks remailers are important for providing for return traffic, but they must be sent to every correspondent individually. In

addition, using a reply block requires that a correspondent be familiar with the use of specialized software. This problem is addressed by nymservers, which act as holding and processing centers for reply blocks.
To use a nymserver, a user simply registers an e-mail address of the form "nym@nymserver.net" and associates a reply block with it. This association can be carried out via anonymous e-mail. Then whenever a message is sent to "nym@nymserver.net", the nymserver automatically prepends the associated reply block, encrypts the aggregate, and sends it off to the appropriate anonymous remailer.
The most popular nymserver may be the one run at nym.alias.net, which is hosted at MIT's Lab for Computer Science. A recent report by Mazieres and Kaashoek details the technical and social details of running the nymserver, including problems of abuse [20].

3.3.5 Remailer User Interfaces

The major reason for the massive popularity of anon.penet.fi was that it was extremely easy to use. Anyone who could type "Request-Remailing-To:" at the top of an e-mail message could send anonymous e-mail. With the advent of remailers which required the use of PGP or the Mixmaster software, the difficulty of using remailers increased. This difficulty was aggravated by the fact that for years, both PGP and Mixmaster were only available as command-line applications with a bewildering array of options.

3.4 Recent Mix-net Designs

3.4.1 Rewebber

Goldberg and Wagner applied Mixes to the task of designing an anonymous publishing network called Rewebber. Rewebber uses URLs which contain the name of a Rewebber server and a packet of encrypted information. When typed into a web browser, the URL sends the browser to the Rewebber server, whch decrypts the associated packet to find the address of either another Rewebber server or a legitimate web site. In this way, web sites can publish content without revealing their location.
Mapping between intelligible names and Rewebber URLs is performed by a name server called the Temporary Autonomous Zone(TAZ), named after a novel by Hakim Bey. The point of the "Temporary" in the name of the nameserver (and

the novel) is that static structures are vulnerable to attack. Continually refreshing the Rewebber URL makes it harder for an adversary to gain information about the server to which it refers [13].

3.4.2 Babel

Contemporary with Cottrell's Mixmaster is Babel, which uses a modified version of PGP as its underlying encryption engine. This modified version does not include normal headers, which would include the identity of the receiver, the PGP version number, and other identifying information.
The Babel paper defines quantities called the "gues factor" and the "mix factor" which model the ability of an adversary to match messages passing through the mix with their original senders. Then several attacks are presented, including the trickle and flooding attack, along with some countermeasures. The paper is noteworthy in that it attempts to give an analysis of just how much the practice of batching messages helps the untraceability of a mix-net node [14].

3.4.3 Stop And Go Mixes

The next step in probabilistic analysis for mix-nets comes in the work of Kesdogan, Egner, and Buschkes, who proposed the "Stop and Go Mix". They divide networks into two kinds: "closed" networks, in which the number of users is small, known in advance, and all users can be made distinct, and "open" networks like the Internet with extremely large numbers of users. They claim that perfect anonymity cannot be achieved in these open networks, because there is no guarantee that every single client of the mix node is not the same person coming under different names. Instead, they define and consider a notion of probabilistic anonymity: given that the adversary controls some percentage of the clients, some other set of mix servers, and is watching a Mix, can the probability of correlating messages be quantified in terms of some security parameter? They consider queueing theory as an inspiration for a statistical model and manage to prove theorems about the adversary's knowledge in this model [17].

3.4.4 Variable Implicit Addresses

Later, Kesdogan et. al. applied Mixes to the GSM mobile telephone setting. Here, the point is to allow for GSM roaming from cell to cell while still protecting

the user's real location from discovery by the phone company or an outside intruder. This is done by the use of variable implicit addresses, which work as follows : each roaming area has a publically known and static explicit address. When the client GSM phone comes online or crosses the boundaries of a cell, it queries the surrounding cells and downloads these addresses. Then it creates a new address for itself which combines the addresses of its surrounding cells.
Then, instead of sending the entirety of the new address, the phone sends only some characters, say log n, of the address to the network to identify itself. The network then directs traffic intended for the phone to any cell which has those log n characters in its address. A refinement process then takes place in which the phone gives out slightly more information to the system to improve performance by sending information to fewer cells, but not so much as to allow its location to be restricted to only one cell.

3.4.5 Jacobsson's Practical Mix

At EUROCRYPT '98, Jakobsson proposed a mix-net which was both practical and could be proved to mix correctly as long as less than 1/2 of the servers were corrupted. The crucial idea is to treat the mixing as a secure multiparty computation in which each party is collaborating to make the collective mix look like a "random enough" permutation on a batch of messages. Then techniques of zero-knowledge proof are used by which each server can prove to all other servers that they are in fact conforming to the mix protocol. Deviating servers cannot produce valid proofs, and so can be caught and excluded from future mixing. Jakobsson's original protocol requires in the neighborhood of 160 modular exponentiations per message per server.
At PODC '99, Jakobsson showed how the use of precomputation could reduce the cost even further. This new "flash mix" required only around 160 modular multiplications per message per server. This level of efficiency makes flash mixing competitive with the encryption used in anonymous remailers, and a serious candidate for low-latency mixing.

3.4.6 Universally verifiable Mix-nets

With Jakobsson's design, the correctness of a mix-net can only be verified by the mix servers themselves. When more than a threshold of servers is corrupt, the verification fails. Because a user of the mix-net may not be aware of the corruption, this failure may be silent and therefore dangerous. One solution to this problem is a universally verifiable mix-net - a mix-net whose correctness can be verified by anyone, regardless of their status as server or user.

The concept was introduced by Killian, and recently a design of this type was proposed at EUROCRYPT '98 by Abe. This design works along the similar broad lines as the Jakobsson design; each mix server uses zero-knowledge proofs to prove that it is acting in accordance with some protocol to randomly mix messages. The difference here is that these proofs are posted publically by the mix nodes instead of being multicast only to other mix nodes. The novel feature of Abe's design is that the work necessary to verify these proofs grows in a fashion independent of the number of servers. Unfortunately, verifying these proofs requires on the order of 1600 modular exponentiations per message.

3.4.7 Onion Routing

The Onion Routing system designed by Syverson, et. al. creates a mix-net for TCP/IP connections. In the Onion Routing system, a mix-net packet, or "onion", is created by successively encrypting a packet with the public keys of several mix servers, or "onion" routers.
When a user places a message into the system, an "onion proxy" determines a route through the anonymous network and onion encrypts the message accordingly. Each onion router which receives the message peels the topmost layer, as normal, then adds some key seed material to be used to generate keys for the anonymous communication. As usual, the changing nature of the onion - the "peeling" process - stops message coding attacks. Onions are numbered and have expire times, to stop replay attacks. Onion routers maintain network topology by communicating with neighbors, using this information to initially build routes when messages are funneled into the system. By this process, routers also establish shared DES keys for link encryption.
The routing is performed on the application layer of onion proxies, the path between proxies dependent upon the underlying IP network. Therefore, this type of system is comparable to loose source routing.
Onion Routing is mainly used for sender-anonymous communications with non-anonymous receivers. Users may wish to Web browse, send email, or use applications such as rlogin. In most of these real-time applications, the user supplies the destination hostname/port or IP address/port. Therefore, this system only provides receiver-anonymity from a third-party, not from the sender.
Furthermore, Onion Routing makes no attempt to stop timing attacks using traffic analysis at the network endpoints. They assume that the routing infrastructure is uniformly busy, thus making passive intra-network timing difficult. However, the network might not be statistically uniformly busy, and attackers can tell if two parties are communicating via increased traffic at their respective endpoints. This endpoint-linkable timing attack remains a difficulty for all low-latency networks [34].

3.4.8 Zero Knowledge Systems

Recently, the Canadian company Zero Knowledge Systems has begun the process of building the first mix-net operated for profit, known as Freedom. They have deployed two major systems, one for e-mail and another for TCP/IP. The e-mail system is broadly similar to Mixmaster, and the TCP/IP system similar to Onion Routing.

ZKS's "Freedom 1.0" application is designed to allow users to use a nym to anonymously access web pages, use IRC, etc [1]. The anonymity comes from two aspects: first of all, ZKS maintains what it calls the Freedom Network, which is a series of nodes which route traffic amongst themselves in order to hide the origin and destination of packets, using the normal layered encryption mix-net mechanism. All packets are of the same size. The second aspect of anonymity comes from the fact that clients purchase "tokens" from ZKS, and exchange these token for nyms - supposedly even ZKS isn't able to correlate identities with their use of their nyms.

The Freedom Network looks like it does a good job of actually demonstrating an anonymous mix-net that functions in real-time. The system differs from Onion Routing in several ways.

First of all, the system maintains Network Information Query and Status Servers, which are databases which provide network topology, status, and ratings information. Nodes also query the key servers every hour to maintain fresh public keys for other nodes, then undergo authenticated Diffie-Hellman key exchange to allow link encryption. This system differs from online inter-node querying that occurs with Onion Routing. Combined with centralized nym servers, time synchronization, and key update/query servers, the Freedom Network is not fully decentralized.

Second, the system does not assume uniform traffic distribution, but instead uses a basic "heartbeat" function that limits the amount of inter-node communication. Link padding, cover traffic, and a more robust traffic-shaping algorithm have been planned and discussed, but are currently disabled due to engineering difficulty and load on the servers. ZKS recognizes that statistical traffic analysis is possible.

Third, Freedom looses anonymity for the primary reason that it is a commercial network operated for profit. Users must purchase the nyms used in pseudonymous communications. Purchasing is performed out-of-band via an online Web store, through credit-card or cash payments. ZKS uses a protocol of issuing serial numbers, which are reclaimed for nym tokens, which in turn are used to anonymously purchase nyms. However, this system relies on "trusted third party" security: the user must trust that ZKS is not logging IP information or recording serial-token exchanges that would allow them to correlate nyms to users. The future adoption of anonymous ecash purchasing should remove this weakness, and allow truely anonymous nym issuing.

3.4.9 Web Mixes

Another more recent effort to apply a mix network to web browsing is due to Federrath et. al. who call their system, appropriately enough, "Web Mixes". From Chaum's mix model, similar to other real-time systems, they use: layered public-key encryption, prevention of replay, constant message length within a certain time period, and reordering outgoing messages. The Web Mixes system incorporates several new concepts. First, they use an adaptive "chop-and-slice" algorithm that adjusts the length used for all messages between time periods according to the amount of network traffic. Second, dummy messages are sent from user clients as long as the clients are connected to the Mix network. This cover traffic makes it harder for an adversary to perform traffic analysis and determine when a user sends an anonymous message, although the adversary can still tell when a client is connected to the mix-net. Third, Web Mixes attempt to restrict insider and outsider flooding attacks by limited either available bandwidth or the number of used time slices for each user. To do this, users are issued a number of blind signature tickets for each time slice, which are spent to send anonymous messages. Lastly, this effort includes an attempt to build a statistical model which characterizes the knowledge of an adversary attempting to perform traffic analysis [3].

3.5 Other Anonymous Channels

3.5.1 The Dining Cryptographers

The Dining Cryptographers protocol was introduced by David Chaum and later improved by Pfitzmann and Waidner as a means of guaranteeing untraceability for the sender and receiver of a message, even against a computationally all-powerful adversary. The protocol converts any broadcast channel into an anonymous broadcast channel. Though there are problems with the efficiency of the protocol and the difficulty of correct implementation, which is why it is not popular [7].

3.5.2 Crowds

The Crowds system was proposed and implemented by AT&T Research, named for collections of users that are used to achieve partial anonymity for Web browsing. A user initially joins some crowd and her system begins acting as a node, or

anonymous jondo, within that crowd. In order to instantiate communications, the user creates some path through the crowd by a random-walk of jondos, in which each jondo has some small probability of sending the actual http request to the end server. Once established, this path remains static as long as the user remains a member of that crowd. The Crowds system does not use dynamic path creation so that colluding crowd eavesdroppers are not able to probabilistically determine the initiator (i.e., the actual sender) of requests, given repeated requests through a crowd. The jondos in a given path also share a secret path key, such that local listeners, not part of the path, only see an encrypted end server address until the request is finally sent off. The Crowds system also includes some optimizations to handle timing attacks against repeated requests, as certain HTML tags cause browsers to automatically issue re-requests.
Similar to other real-time anonymous communication channels (Onion Routing, the Freedom Network, Web Mixes), Crowds is used for senders to communicate with a known destination. The system attempts to achieve sender-anonymity from the receiver and a third-party adversary. Receiver-anonymity is only meant to be kept from adversaries, not from the sender herself.
The Crowds system serves primarily to achieve sender and receiver anonymity from an attacker, not provide unlinkability between the two agents. Due to high availability of data - real-time access is faster than mix-nets as Crowds does not use public key encryption - an adversary can more easily use traffic analysis or timing attacks. However, Crowds differs from all other systems we have discussed, as users are members of the communications channel, rather than merely communicating through it. Sender-anonymity is still lost to a local eavesdropper that can observe all communications to and from a node. However, other colluding jondos along the sender's path - even the first-hop - cannot expose the sender as originated the message. Reiter and Rubin show that as the number of crowd members goes to infinity, the probable innocence of the last-hop being the sender approaches one [31].

3.5.3 Ostrovsky's Anonymous Broadcast via XOR-Trees

In CRYPTO '97, Ostrovsky considered a slightly different model of anonymous broadcast. In this model, there are n servers broadcasting into a shared broadcast channel. One of the servers is a special "Command and Control" server; the rest are broadcasting dummy traffic. Then there is an adversary who has control of some of the servers and wants to know which server is the "Command and Control". Ostrovsky shows how to use correlated pseudo-random number generators whose output reveals a certain message when XORed together to create a protocol which prevents the adversary from discovering which server is the correct one, even if he can eavesdrop on all communications and corrupt up to k servers, where k is a security parameter which affects the efficiency of the

protocol [9].

3.6 Summary

Based on the study of state of art in anonymous communications we now know that Proxies only provide unlinkability between the sender and the receiver, given that the proxy is not compromised. It is an easy task to intersect attacks and make traffic analysis if an adversary monitor traffic to the proxy.
In the Chaumian mix-nets a chain of mix-nets are used to provide anonymity. Each mix in the chain strips off the identifying marks on incoming messages and then sends the message to the next Mix, based on routing instructions which is encrypted with its public key. The problem with this design is that it is based on a static network. In a dynamic environment it is easy to make traffic analysis, because of the easy identification of the Mixes. But the basic workings of a Mix-net is useable for our project. We only need to develop a more dynamic Mix selection technique, and design a more suitable method of routing between Mix nodes for a highly dynamical environment.
In the next chapter the state of art protocols that are developed for MANETS are presented. We will investigate pros and cons for the various protocols to finally develop our own protocol.

Chapter 4

Recent Anonymity Designs in MANET's

In this chapter the start of art in anonymous communications in MANET's is presented. There are three papers which is being reviewed. There is given a thorough presentation, which will later lead to the analysis and design of our own protocol.

4.1 ANODR

The purpose of the paper of Hong and Kong [18] (called ANODR) is to develop "untraceable" routes or packet flows in an on-demand routing environment. This goal is very different from other related routing security problems such as resistance to route disruption or prevention of "denial-of-service" attacks. In fact, in ANODR the enemy will avoid such aggressive schemes, in the attempt to be as "invisible" as possible, until it traces, locates, and then physically destroys the assets. They address the untraceable routing problem by a route pseudonymity approach. In their design, the anonymous route discovery process establishes an on-demand route between a source and its destination. Each hop on route is associated with a random *route pseudonym*. Since data forwarding in the network is based on route pseudonyms with negligible overhead, local senders and re-

ceivers need not reveal their identities in wireless transmission. In other words, the route pseudonymity approach allows to "unlink" (i.e., prevent interference between) network member's location and identity. For each route, they also ensure unlinkability among its route pseudonyms. As a result, in each locality eavesdroppers or any bystander other than the forwarding node can only detect the transmission of wireless packets stamped with random route pseudonyms. It is hard for them to trace how many nodes in the locality, who is the transmitter or receiver, where a packet flow comes from and where it gores to (i.e., what are the previous hops and the next hops on route), let alone the source sender and the destination receiver of the flow. They further tackle the problem of node intrusion within the same framework. In their design a strong adversary with node intrusion capability must carry out a complete "vertex cover" process to trace each on-demand ad hoc route.
The design of route pseudonymity is based on a network security concept called "broadcast with trapdoor information". Multicast/broadcast is a network-based mechanism that provides recipient anonymity support to their project. Trapdoor information is a security concept that has been widely used in encryption and authentication schemes. ANODR is realized upon a hybrid form of these two concepts.
Their contributions is to present an *untraceable and intrusion tolerant routing protocol* for mobile ad hoc networks.

- *Untraceability*: ANODR dissociates ad hoc routing from the design of network member's identity/pseudonym. The enemy can neither link network member's identities with their locations, nor follow a packet flow to its source and destination. Though the adversaries may detect the existence of local wireless transmissions, it is hard for them to infer a covert mission's number of participants, as well as the transmission pattern and motion pattern of these participants.

- *Intrusion tolerance*: ANODR ensures there is no single point of compromise in ad hoc routing. Node intrusion does not compromise location privacy of other legitimate members, and an on demand ANODR route is traceable only if all forwarding nodes on route are intruded.

In their paper they address the anonymous routing problem especially as Pfitzmann and Köhntopp [26] who define the concept of *pseudonymity* and the concept of *anonymity* in terms of *unlinkability* or *unobservability*.
In a computer network, entities are identified by unique IDs. Network transmissions are treated as the *Items Of Interest* (IOIs). *Pseudonym* is an identifier of subjects to be protected. It could be associated with a sender, a recipient, or any entity demanding protection. The concept of *pseudonymity* is defined as the use of pseudonyms as IDs. The concept of *anonymity* is defined in terms of

either *unlinkability* or *unobservability*. The difference between unlinkability and unobservability is whether security protection covers IOIs or not:

- *Unlinkability*: Anonymity in terms of unlinkability is defined as unlinkability of an IOI and a pseudonym. An anonymous IOI is not linkable to any pseudonym, and an anonymous pseudonym is not linkable to any IOI. More specifically, *sender anonymity* means that a particular transmission is not linkable to any sender's pseudonym, and any transmission is not linkable to a particular sender's pseudonym. *Recipient anonymity* is similarly defined.
  A property weaker than these two cases is *relationship anonymity* where two or more pseudonyms are unlinkable. In particular for senders and recipients, it is not possible to trace who communicates with whom, though it may be possible to trace who is the sender, or who is the recipient. In other words, sender's pseudonym and recipient's pseudonym (or recipient' pseudonyms in case of multicast) are unlinkable.

- *Unobservability*: Unobservability also protects IOIs from being exposed. That is, the message transmission is not discernible from random noise. More specifically, *sender unobservability* means that a could-be sender's transmission is not noticeable. *Relationship observability* means that it is not noticeable whether anything is sent from a set of could-be senders to a set of could-be recipients.

Throughout their paper, IOI means wireless transmission in mobile ad hoc networks. They use the term "anonymity" as a synonym of "anonymity in terms of unlinkability". In other words, they do not address how to make wireless transmissions indistinguishable from random noises, thus unobservability is not studied in their project. Instead, they address two closely-related unlinkability problems for mobile ad hoc networks.
They study the *route anonymity* problem to implement an untraceable routing scheme, where each route consists of a set of hops and each hop is identified by a route pseudonym. For each mulit-hop route, they seek to realize relationship anonymity among the corresponding set of route pseudonyms. The route pseudonymity approach differentiates their work from earlier studies addressing identity pseudonymity (e.g., person pseudonymity, role pseudonymity, and transaction pseudonymity).
The route pseudonymity approach enables *location privacy* support that realizes unlinkability between a mobile node's identity and its location. This is achieved by anonymous wireless communications that hide the sender and receiver. This part covers the traditional meaning of sender anonymity, recipient anonymity, and relationship anonymity in a wireless neighborhood.

4.1.1 Design of ANODR

ANODR divides the routing process into two parts: *anonymous route discovery* and *anonymous route maintenance*. Besides, in *anonymous data forwarding* data packets are routed anonymously from senders to receivers as usual. The details of these parts are described below:

4.1.1.1 Anonymous route discovery

Anonymous route discovery is a critical procedure that establishes random route pseudonyms for an on-demand route. A communication source initiates the route discovery procedure by assembling an RREQ packet and locally broadcasting it. The RREQ packet is of the format

$$< RREQ, seqnum, tr_{dest}, onion >$$

where (i) *seqnum* is a globally unique sequence number. (ii) tr_{dest} is a cryptographic trapdoor that can only be opened by the destination. Depending on the network's cryptographic assumptions, how to realize the global trapdoor is an implementation-defined cryptographic issue. (iii) *onion* is a cryptographic onion that is critical for route pseudonym establishment.
Using cryptographic onion in RREQ network-wide flooding raises design validity concerns as well as performance concerns. There are presented three variants to illustrate their design. The first one is a naive porting of mix-net to mobile ad hoc networks. The last one features best anonymity guarantee and best performance.
Like mix-net, the cryptographic onion is formed as a public key protected onion (PO). The corresponding ANODR-PO protocol is described below.

1. RREQ phase: RREQ packets with previously seen sequence numbers are discarded. Otherwise, as depicted in Figure 4.1, each RREQ forwarding node X prepends the incoming hop to the PO structure, encrypts the result with its own public key PK_X, then broadcasts the RREQ locally.

2. RREP phase: When the destination receives an RREQ packet, the embedded PO structure is a valid onion to establish an anonymous route towards the source. The destination opens the trapdoor and assembles an RREP packet of the format

   $$< RREP, N, pr_{dest}, onion >$$

   where *onion* is the same cryptographic onion in the received RREQ packet, pr_{dest} is the proof of global trapdoor opening, and N is a locally unique

random route pseudonym. The RREP packet is then transmitted by local broadcast. Unlike RREQ phase when the as hoc route is determined, the RREP phase is less time-critical and is implemented by reliable transmissions. As depicted in Figure 4.1, any receiving node X decrypts the onion using its own private key SK_X. If its own identity pseudonym X does not match the first field of the decrypted result, it then discards the packet. Otherwise, the node is on the anonymous route. It selects a locally unique nonce N', stores the correspondence between $N \rightleftharpoons N'$ in its forwarding table, peels off one layer of the onion, replaces N with N', then locally broadcasts the modified RREP packet. The same actions will be repeated until the source receives the onion it originally sent out.
Upon receiving different RREQ packets, the destination can initiate the same RREP procedure to realize multiple anonymous paths between itself and the source.

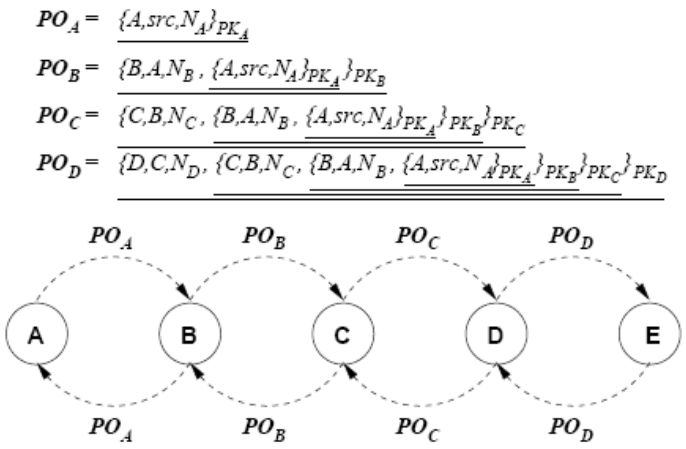

Figure 4.1: ANODR-PO: Anonymous route discovery using public key cryptographic (A single path showed from source A to destination E)

Firstly, this ANODR-PO scheme has a significant drawback. As RREQ is a network-wide flooding process, large processing overhead will exhaust computation resources at the entire network level.
The efficient anonymous route discovery protocol is depicted in Figure 4.2. Instead of relying on public key encrypted onions, the new scheme ANODR-BO uses symmetric key based *Boomerang Onions*(BO).

1. When intermediate forwarding node X sees an RREQ packet, it prepends

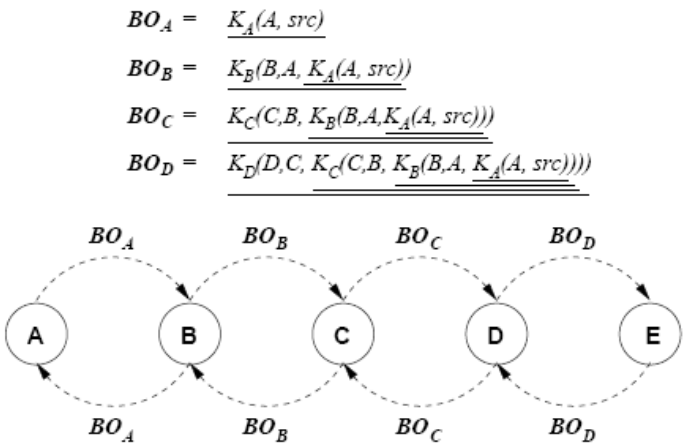

Figure 4.2: ANODR-BO: Anonymous route discovery using Boomerang Onion (A single path showed from source A to destination E)

the incoming hop to the boomerang onion, encrypts the result with a random symmetric key K_X, then broadcasts the RREQ locally.

2. The boomerang onion will be bounced back by the destination. Like the public key version, when node X sees an RREP packet, it strips a layer of the boomerang onion and locally broadcasts the modified RREP packet. Finally the source will receive the boomerang onion it originally sent out.

Compared to ANODR-PO, ANODR-BO ensures that no public key operation is executed during RREQ flooding, hence the impact on processing latency is acceptable because many symmetric key encryption schemes have good performance even on low-end devices.
Secondly, ensuring identity anonymity for ad hoc network members is i critical design goal. Figure 4.3 shows the case where anonymous route discovery depends completely on local broadcast with trapdoor information. The depicted ANODR-TBO only uses trapdoor boomerang onions (TBO).

1. When intermediate forwarding node X sees an RREQ packet, it embeds a random nonce N_X to the boomerang onion (this nonce is not a route pseudonym nonce), encrypts the result with a random symmetric key K_X, then broadcasts the RREQ locally. The trapdoor information consists of N_X and K_X, and is only known to X.

2. The boomerang onion will be bounced back by the destination. After each

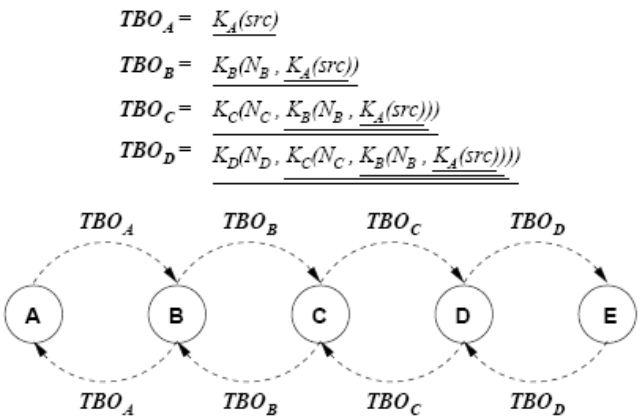

Figure 4.3: ANODR-TBO: Anonymous route discovery using Trapdoor Boomerang Onion (A single path showed from source A to destination E)

local RREP broadcast, only the next hop (i.e., the previous hop in RREQ phase) can correctly open the trapdoor it made in the RREQ phase, hence the result is equivalent to a wireless unicast. Then the node strips a layer of the boomerang onion and locally broadcasts the modified RREP packet.

4.1.1.2 Anonymous data forwarding

For each end-to-end connection, the source wraps its data packets using the outgoing route pseudonym in its forwarding table. A data packet is then broadcast locally without identifying the sender and the local receiver. The sender does not bother to react to the packet it just sent out. All other local receiving nodes must look up the route pseudonym in their forwarding tables. The node discards the packet if no match is returned. Otherwise, it changes the route pseudonym to the matched outgoing pseudonym, then broadcasts the changed data packet locally. The procedure is then repeated until the data packet arrives at the destination.

4.1.1.3 Anonymous route maintenance

Following the soft state design the routing table entries are recycled upon timeout T_{win}. Moreover, when one or more hop is broken due to mobility or node

failures nodes cannot forward packet via the broken hops. Upon anomaly detection, a node looks up the corresponding entry in its forwarding table, finds the other route pseudonym N' which is associated with the pseudonym N of the broke hop, and assembles a route error packet of the format $< RERR, N' >$. The node then recycles the table entry and locally broadcasts the RERR packet. If multiple routes are using the broken hop, then each of them will be processed and multiple RERR packets are broadcast locally.
A receiving node of the RERR packet looks up N' in its forwarding table. If the lookup returns result, then the node is on the broken route. It should find the matched N'' and follow the same procedure to notify its neighbors.

4.1.2 Summary

In the paper of ANODR they propose an anonymous on-demand routing protocol for mobile ad hoc networks. They addressed two close-related unlinkability problems, namely *route anonymity* and *location privacy*. Based on a route pseudonymity approach, ANODR prevents adversaries, such as node intruders and omnipresent eavesdroppers, from exposing local wireless transmitters' identities and tracing ad hoc network packet flows. The design of ANODR is based on "broadcast with trapdoor information", a novel network security concept with hybrid features merged from both network concept "broadcast" and security concept "trapdoor information". This network security concept can be applied to multicast communication as well.

4.2 Mobility Changes Anonymity

The purpose of Hong and Kong's paper [15] is to identify new anonymity requirements for mobile wireless networks. Their study has two folds: (1) to show that mobility has changed the underlying assumption of existing anonymity research, thus mobile anonymity cannot be ensured by existing proposals designed for fixed networks; (2) they study design principles of new countermeasures. For mobile wireless networks, their study suggests that a hybrid approach of identity-free routing and on-demand routing provides better anonymity support than other approaches. The contributions of their study are listed below:

- They show that anonymity research in fixed networks does not address the new threats that they identified. Since mobility dissociates node identities from a topological or physical location, now mobile nodes need more anonymity supports to protect their location privacy and to hide their motion patterns. Various anonymity attacks studied in their paper effectively break existing anonymity schemes designed for fixed networks.
- Given a reasonable assumption that adequate physical protection is not feasible for all mobile nodes, they argue that identity-free routing is needed to hide a node's identity from its neighboring forwarders. In addition, since mix-net and proactive routing approach are vulnerable to single point of compromise if used in mobile wireless networks, they also show that on-demand routing is a better approach to protect mobile wireless networks.

They study various new anonymity threats in mobile ad hoc networks. They limit their research scope in network layer routing. In other words, anonymity problems at the physical layer or the application layer are not studied in their paper.

4.2.1 Differentiate identity anonymity and venue anonymity

Figure 4.4 illustrates an adversary's network which is comprised of a number of eavesdropping nodes. Each node corresponds to a vertex in an undirected graph $G =< V, E >$, where adversarial eavesdropping nodes form a vertex/$venue^2$ set V , and topological links amongst the nodes form an edge set E. This grid structure demonstrates several possible attacks. On one hand, it characterizes the capability of a collection of collaborative traffic analysts from multiple nodes. On the other hand, it also characterizes the capability of a mobile traffic analyst traveling along the grids to launch anonymity attacks anywhere and anytime.

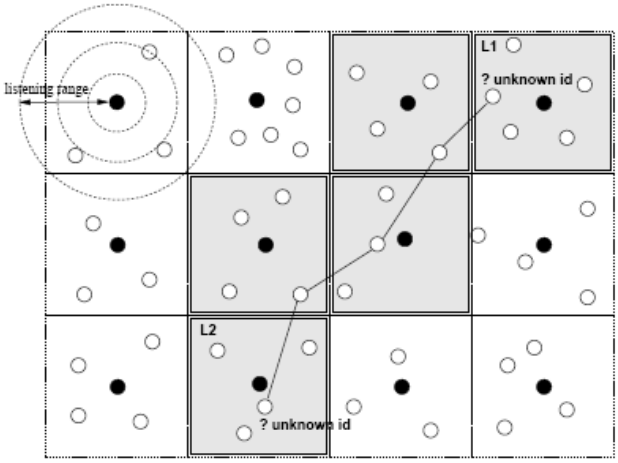

Figure 4.4: Underlying graph $G =< V, E >$ (Traffic analysts are depicted as solid black nodes. A sender in cell $L1$ is communicating with a recipient in cell $L2$. Identified active routing cells are depicted in shade.)

They argue that in fixed networks, a sender (or recipient) and its venue are synonyms, that is, identifying a sender's (or recipient's) venue implies the compromise of sender (or recipient) anonymity. But in mobile networks, a node's identity is dissociated from a specific venue. However, at each traffic analyst's vertex/venue, the adversarial analyst can correlate node identities with its own exact location (e.g., obtained via a positioning system like GPS). In example 1, 2 and 3 Hong and Kong show that identity anonymity and venue anonymity are different concepts in mobile networks. While identity anonymity is still an issue, venue anonymity is a new problem that should be addressed separately. In particular, the new venue anonymity set is comprised of all vertexes/venues, and the sender/recipient venue should not be identifiable within the new anonymity set given all intercepted IOIs.

4.2.1.1 Example 1

(Sender or recipient identity anonymity attack in on-demand route request flooding) *Hong and Kong argue that in common on-demand ad hoc routing schemes like DSR and AODV, identities of the source/sender and the destination/recipient are explicitly embedded in route request (RREQ) packets. Any external adversary who has intercepted such a flooded packet can*

uniquely identify the sender's and the recipient's identities, but may not know the venue/vertex of the sender or the recipient.

4.2.1.2 Example 2

(Per-hop encryption may not protect sender or recipient identity anonymity against internal adversary) *A seemingly-ideal cryptographic protection is to apply pairwise key agreement on every single hop, so that a single-hop transmission is protected by an ideal point-to-point secure channel between the two ends of the hop. The secure channel also protects every packet including the packet header. This solution prevents external adversary from understanding routing messages and network topology, but unfortunately does not prevent any internal DSR/AODV network member from identifying the sender's and the recipient's identities upon receiving a flooded RREQ packet.*

4.2.1.3 Example 3

(Packet flow tracing attack) *This attack reveals the relationship between a sender's venue and its recipient's venue. On a (multi-hop) forwarding path, timing correlation and content correlation analysis can be used to trace a packet flow. (1) Timing correlation analysis: The adversary can use timing information between successive transmission events to trace a victim message's forwarding path. With no background traffic, a packet forwarded to node X at time t and a packet forwarded from the same node at time $(t + O)$ are very likely on the same packet flow. (2) Content correlation analysis: A control/data flow can be traced by content correlation (e.g., comparing data field contents and length amongst local transmissions). In Figure 4.4, collaborative adversarial analysts can trace an ongoing packet flow to the sender's venue $L1$ and the recipient's venue $L2$, thus break sender (or recipient) venue anonymity. But they may not be able to identify the sender's (or recipient's) identity. Hong and Kong say that this is possible in ANODR [18] where routing is completely free of sender's and recipient's identities.*

4.2.2 Privacy of location and motion pattern

In fixed networks, a fixed node's topological location and related physical location are determined a priori. Besides, the motion pattern of a fixed node is not a network security concern. In other words, there is no need to ensure privacy

for a network node's location and motion pattern. Therefore, in anonymity solutions proposed for fixed networks, a network node is allowed to know its neighborhood. For example, a Chaumian mix knows its immediate upstream and downstream mixes, a jondo in Crowds knows its next jondo or the destination recipient. If directly ported from the fixed networks, these schemes do not ensure location privacy near any internal adversary, which can launch attacks described in Example 4.

4.2.2.1 Example 4

(One-hop location privacy attack) *Given any cell L depicted in Figure 4.4, the inside wireless traffic analyst may gather and quantify (approximate) information about active mobile nodes, for example, (a) enumerate the set of currently active nodes in L; (b) related quantities such as the size of the set; (c) traffic analysis against L, e.g., how many and what kind of connections in-and-out the cell.*

Ensuring privacy for mobile nodes motion pattern is a new expression. Example 5 gives a brief overview of the attack. If the network fails to ensure one-hop location privacy, they have showed that a mobile node's motion pattern privacy can be compromised by a dense grid of traffic analysts, or even by a sparse set of internal adversarial nodes under certain conditions, for example, when (1) a node is capable of knowing neighbors' relative positions (clockwise or counter-clockwise), and (2) in DSR/AODV's on demand route discovery, RREP traffic of the same source/destination pair is correlatable.

4.2.2.2 Example 5

(Motion pattern inference attack) *As implied by the name, the goal of this passive attack is to infer (possibly imprecise) motion pattern of mobile nodes. For example, collaborative adversaries can monitor wireless transmissions in and out a specific mobile node, they can combine the intercepted data and trace the motion pattern of the node. In some cases, a network mission may require a set of legitimate nodes to move towards the same direction or a specific spot. Motion pattern inference attack can effectively visualize the outline of the mission. In a network with dense adversarial analysts, motion pattern inference attack can be trivially implemented on top of one-hop location privacy attack using stored historical records.*

Mobile networks could be deployed in severe environments, where nodes with inadequate physical protection are susceptible to being captured and compromised. Any node in such a network must be prepared to operate in a mode

that allows no gullibility. In the network, the combination of infrastructureless networking and location privacy presents a dilemma described in Example 6.

4.2.2.3 Example 6

(Location privacy dilemma in infrastructureless networks with internal adversary) *In mobile routing schemes without infrastructure support, a node must rely on at least one of its neighbors to forward its packets. When anonymity service is concerned, a node is facing a dilemma. On one hand, it must forward its packets to one of its neighbors, so that the neighbor(s) can further forward the packets towards the destination. On the other hand, the node does not know whether there is an adversarial node amongst its neighbors, and if yes, which neighbor is adversarial. This dilemma calls for identity-free routing that does not reveal a node's identity information to its neighbors.*

4.2.3 Privacy of ad hoc network topology

In a fixed network, network topology is physically determined a priori. Hence there is no such difference (and associated privacy concerns). However, in mobile networks network topology constantly changes due to mobility. Once the adversary knows fresh network topology, it can break the network's anonymity protection given other out-of-band information like geographic positions and physical boundaries of the underlying mobile network. Privacy of network topology becomes a new anonymity aspect in mobile networks.
In fixed Internet, proactive routing schemes like BGP, OSPF and RIP are widely used in inter-domain routing and intra-domain routing. Every router possesses abundant knowledge about network topology if the underlying routing scheme is hierarchical, or complete knowledge about the entire network topology if the underlying routing scheme is flat. This is not a problem for the fixed Internet. In proactive ad hoc routing protocols like DSDV, OLSR and TBRPF, mobile nodes also constantly exchange routing messages, so that each sender node knows enough network topological information to find any intended recipient. In a typical network with pairwise end-to-end communication pattern, this means at each moment every sender node knows abundant network topological information about all other nodes. Thus a single adversarial sender can break anonymity protection of the underlying mobile network. This remark is justified in the following Example 7 and 8.

4.2.3.1 Example 7

(A compromised sender tries to locate where a specific node is) *An anonymous routing protocol should prevent a sender from knowing a (multi-hop) forwarding path towards any specific mobile node. Otherwise, a compromised network member can simply function as a sender to trace any mobile node at its convenience. This example shows that pre-computed routing schemes, in particular proactive routing schemes that accumulate a posteriori network topology knowledge on each sender, directly conflicts with anonymity protection in mobile networks. Any equivalence of proactive routing scheme, such as enforcing requirement to let node send out unsolicited advertisements to other nodes so that network topology can be well-known in the network, also directly conflicts with mobile anonymity protection. The network topology knowledge collected on mobile nodes can be used by the adversary to fight against the network. If node compromise is feasible, such design indeed establishes a lot of single points of compromise in the network.*

4.2.3.2 Example 8

(Vulnerabilities of mix-net in mobile networks) *In mix-net, the entire forwarding path must be determined on the sender prior to anonymous data delivery. Proactive routing schemes may be used in mix-nets to let sender gather the needed network topology knowledge, but this design choice is not resilient to internal threats. If the Chaumian mix-net is directly ported into a mobile network by treating all or some mobile nodes as Chaumian mix nodes, then any adversarial sender knows the entire topology of the mix-net.*

Compared to source routing, link state routing and distance vector routing, virtual circuit based schemes only store information about next link ID for each session. With appropriate design, it is not necessary to reveal a node's identity to neighbors. This identity-free routing strategy minimizes information leakage in spite of node intrusions. On the other hand, compared to proactive schemes, on-demand schemes are less vulnerable to internal threats since they do not require mobile nodes to acquire fresh network topology knowledge. Based on these observations, Hong and Kong believe that a hybrid of identity-free routing and on-demand routing provides better anonymity support in mobile ad hoc networks.

4.2.4 Simulation Study

ANODR is the first identity-free and purely on-demand ad hoc routing protocol proposed. In ANODR, node identities are never used in routing and thus never revealed to adversary. Hong and Kong show how the adoption of various cryptosystems has great impact on anonymous routing performance. They have implemented the following ANODR variants.

1. ANODR, where pairwise key agreement between two consecutive RREP forwarders is implemented by key exchanges using one-time public keys.
2. ANODR-KPS, where the needed key agreement is implemented by Key Pre-distribution Schemes (KPS) instead of public key cryptography. In particular, ANODR-BLOM-KPS uses Blom's deterministic KPS and ANODR-DU-KPS uses Du's probabilistic KPS. In ANODR-DU-KPS, the probability of a successful key agreement per hop is 98%, which means during RREP phase the probability of establishing the anonymous virtual circuit per hop is 98%. With 2% at every hop, key agreement fails and new route discovery procedure must be invoked.

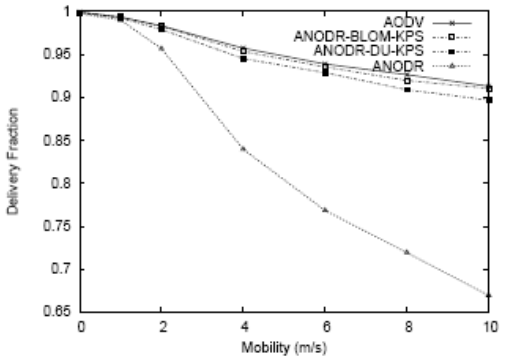

Figure 4.5: Data Packet Delivery Fraction

Figure 4.5 gives the packet delivery fraction as a function of increasing mobility. The figure shows that ANODRKPS's perform almost as good as optimized AODV. This result can be justified by the following reasons: (1) The onion and/or the key agreement material used in ANODR's and/or ANODR-KPSs' control packets, and the route pseudonym field used in data packets are not big enough to incur noticeable impact to the packet delivery fraction. (2) The 0.02ms/1ms cryptographic computation overhead for the two schemes is too

small to make a difference in route discovery. The latter reason also explains why the performance of ANODR degrades faster than ANODR-KPSs - the long encryption/decryption computation time of ANODR prolongs the route acquisition delay, which reduces the accuracy of the newly discovered route, leading to more packet losses. (3) The route optimization of AODV has less effect when a network is at a medium size - 150 nodes. Further, the figure shows that ANODR-DU-KPS has lower delivery ratio than ANODR-BLOM-KPS. The reason for the degradation is the failed probabilistic key agreement along the RREP path. The source only has 0:98k (k is the path length, here, the average is 4-5 hops) chances of receiving a RREP, which may be small for some paths. The source has to initiate a new route discovery in the absence of an expected RREP, resulting in higher control overhead and lower performance. Clearly, the figure shows the tradeoff concern between the performance and the degree of protection.

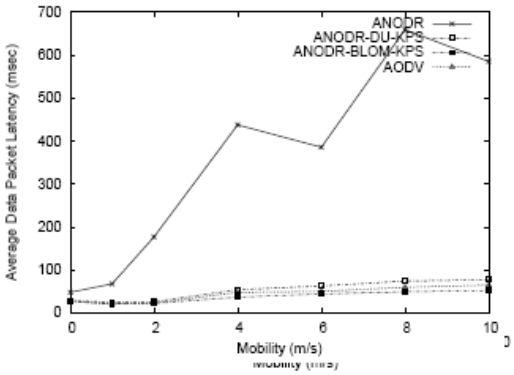

Figure 4.6: End-to-end Data Packet Latency

Figure 4.6 shows the average end-to-end data packet latency when mobility increases. ANODR-KPS's and AODV exhibit very close end-to-end packet latency as they require very small processing time. ANODR has much longer latency than the aforementioned three due to additional public key processing delay during RREP phase. ANODR-DUKPS has a little longer end-to-end packet delay than the other two due to probabilistic failures. The delay trend of all the protocols increases when mobility increases, leading to increasing buffering time in waiting for a new route discovery.

Figure 4.7 gives the number of control bytes being sent in order to deliver a single data byte. All ANODR variants send more control bytes than AODV, because they use larger packets due to global trapdoors, cryptographic onions, and KPS key agreement materials. In particular, either ANODR-KPS uses long key agreement materials. When mobility increases, the lack of optimization in

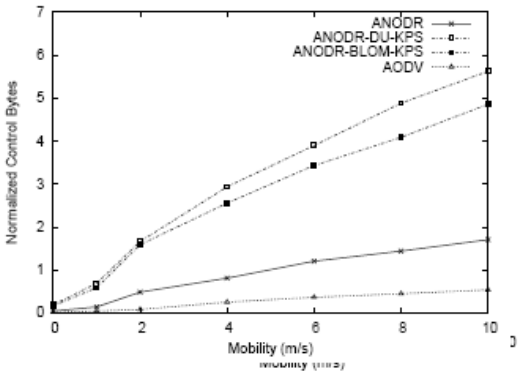

Figure 4.7: Normalized Control Bytes

ANODR variants demonstrates here a faster increasing trend as more recovery are generated from sources.

4.2.5 Summary

In their paper Hong and Kong have studied unique anonymity threats in mobile ad hoc networks. Unlike a fixed network, a mobile ad hoc network should prevent its mobile network members from being traced by passive adversary. The network needs more anonymity protections like (1) venue anonymity in addition to conventional identity anonymity, (2) privacy of node's location and motion pattern, and (3) privacy of ad hoc network topology. Many anonymous schemes designed so far have not considered at least one of these new threats. They use ANODR and its KPS-based variants to show that the efficiency of anonymous routing is an open challenge. ANODR employs on-demand routing and identity-free routing to provide anonymity protection for mobile nodes. Nevertheless, their simulation study shows that routing performance changes significantly when different cryptosystems are used to implement the same function (i.e., pairwise key agreement perhop).

4.3 AD-MIX Protocol

In this section we describe the operations of AD-MIX [33]. Before describing the protocol, some background information and an analogy to aid in understanding the protocol are presented.

4.3.1 Background

The functioning of the AD-MIX protocol is similar in principle to the mix-net approach. AD-MIX adapts the technique employed by mix-nets to encourage participation in wireless ad hoc networks.
AD-MIX promotes participation through information hiding, i.e., none of the nodes through which the packets pass are aware of the ultimate destination of the packet. The destinations of the packets are obscured by using nodes, called *poles*, whose function is similar to the mix servers of the mix-net protocol. Since mixes are used to conceal the true destination of the packets, selfish nodes cannot risk dropping a packet based on the packet's destination address. By dropping packets, they may miss packets potentially destined for them. Therefore, nodes are encouraged to forward all packets passing through them.

4.3.2 Foundations of the AD-MIX Protocol

A node can be classified as a source, destination node, polar node or nonpolar node. Polar nodes, also called poles, are synonymous to the mix servers of the mix-net protocol. The operation of AD-MIX can be explained by describing its operation at the source, non-polar and polar nodes.

4.3.2.1 Operation at the Source Node

When a node wants to transmit a packet, it chooses between 0 and 2 nodes from the network through which the packets should pass. Poles can either be randomly picked or chosen for best performance. The polar nodes chosen need not be neighbors of the source node. They are referred to as *poles*, since they are the extreme points or poles of the packets as the packets propagate towards their destination. The packet to be transmitted is encrypted with the public key of the destination, followed by the public keys of each of the poles in the reverse order of selection. Any public key system such as RSA can be used for

this purpose. The packet is then transmitted with the destination set to the first polar node (or the ultimate destination, if no polar nodes were chosen).
In AD-MIX they impose the condition that no two consecutive poles can be identical. Therefore, if a pole, on decrypting a packet, finds the next destination to be its own address, it will know that it is the ultimate destination of the packet, and that the packet contents are unencrypted. An example is given: Consider the sample network shown in Figure 4.8(a) where a packet is transmitted from source S to destination D through poles M and U. The packet is encrypted three times, first with the public key of D, followed by that of U, and finally with the public key of M. The packet transmitted by S has M as its destination, with the contents of the packet encrypted with public key of M. The packet is sent to M through L. Since the packet is signed with the public key of M, no other intermediate node can interpret the packet contents.

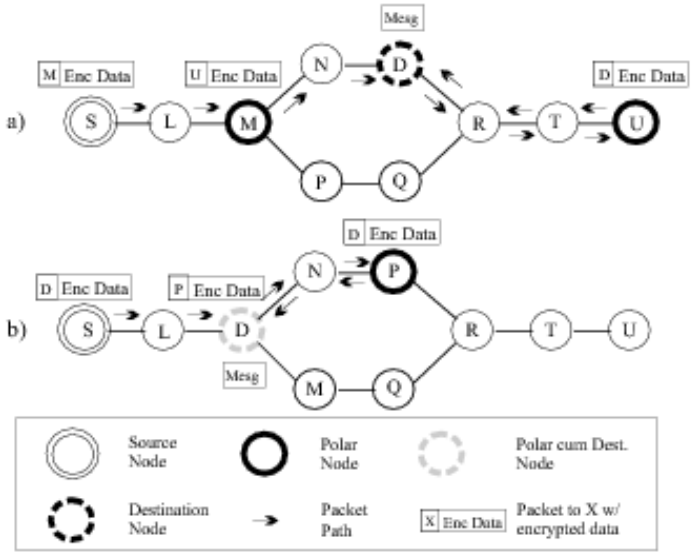

Figure 4.8: The figure shows two scenarios that encourage (a) Non-Polar and (b) Polar nodes to forward packets destined for other nodes. These scenarios are called *loopback*, since the packet loops back to a node that had previously forwarded it.

4.3.2.2 Operation at the Non-Polar Nodes

A non-polar node does not perform any significant function apart from forwarding packets to its neighbor on route to the packet's destination. The possibility of the node being the ultimate destination of the packet provides incentive for the node to forward the packet.
A scenario where a non-polar node forwarding the packet node is the ultimate destination of the packet is depicted in Figure 4.8(a). In the figure, D is a non-polar node that forwards the packet to U, which is merely a pole. The packet, after passing through U, returns back to D. If D, behaving selfishly, had dropped the packet, it would have lost packets destined for itself.

4.3.2.3 Operation at the Polar Nodes

The decrypted packet contains the address of the next pole/destination and some data. If the address of the following pole is the same as the current node's address, then the data contains the unencrypted message to be delivered to this node; this node is the ultimate destination of the packet. otherwise, the next pole is either another pole or the ultimate destination of the packet, and the data contains the packet to be transmitted, encrypted with the public key of the next pole. This packet is transmitted to the next pole.
In the scenario in Figure 4.8(b), where a polar node forwarding the packet ends up as the ultimate destination of the packet. After receiving the packet, polar node D decrypts it with its private key and determines that the packet should be forwarded to P. P, after receiving the packet, decrypts it to find that the packet should be forwarded to D. This decrypted packet contains the packet to be forwarded, encrypted with the public key of D. Finally, after node D decrypts the packet, it discovers that it is the destination of the packet. If D, behaving selfishly, had dropped the packet instead of forwarding it to P, it would have lost its own packet.

4.3.2.4 Operation of AD-MIX

Figure 4.9 shows an example where a packet is transmitted by source S to destination D, with poles M and P. At source S, a dummy header with D as destination is appended to the message *Mesg* to be transmitted to D. This is then encrypted with the public key of D, followed with the public key of M. This is repressented by '1' in the legend of the figure. The packet transmitted by the source has M as its destination. Non-polar node L, after receiving the packet, forwards it to M. Since the packet is encrypted with the public key of

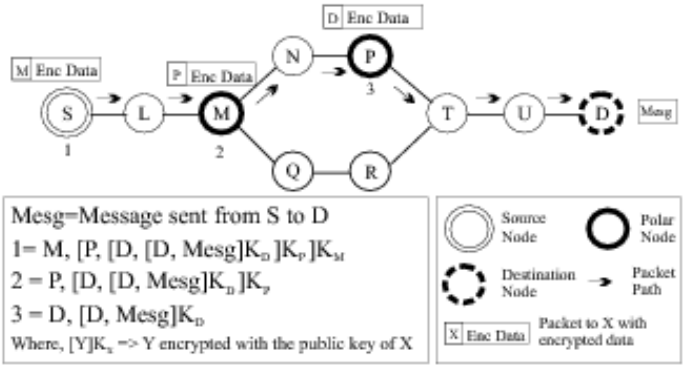

Figure 4.9: Transmission of *Mesg* from node S to node D through poles M and P. The rectangles above the node indicate the packet as seen by the node, while the number below the node corresponds to the actual packet contents as indicated in the legend.

M, L is unable to decrypt its contents. Polar node M, being the intermediate destination of the packet, decrypts it with its public key to find the message to be forwarded to P. The content of this decrypted message is depicted by '2' in the legend. Since this message is encrypted with the public key of P, it cannot be decrypted by M. Polar node M forwards this packet to P, via non-polar node N. Polar node P, after receiving the packet, decrypts it and finds that the packet has to be forwarded to D. The content of the decrypted packet is shown by '3' in the legend. This packet, after being forwarded by non-polar nodes T and U, reaches destination D. Node D, after decrypting the packet, finds the next forwarding address (in the dummy header) to be D again. Hence, it knows that it is the destination of the message.

4.3.3 Other Strategies Employed by AD-MIX

The basic strategy described above is sufficient to encourage participation. They try to improve the efficiency of AD-MIX. These optimizations result in significant improvements to the performance of the protocol.

4.3.3.1 DSR as the Routing Protocol

For an ad hoc network with n nodes running AD-MIX, the worst case hop count of a packet is $3 * (n - 1)$. A packet traversing the worst case path will cause a substantial increase in the delivery time if an on-demand routing protocol is used. Also, choosing poles randomly results in a low probability of sources/poles having a route to the next pole/destination. For reactive routing protocols like AODV and DSR, this may result in up to two additional route discoveries per packet to determine the route to the next pole, and possibly another route discovery to determine the route from the final pole to the destination.
Thus the choice of polar nodes significantly affects the performance of the protocol. It is advantageous to choose poles along the route to the destination so that there is no significant increase in path length. Choosing poles along the route will also result in fewer route discoveries to find the route to the next pole/destination in reactive protocols. Hence, it is advantageous to employ source-routing protocols like DSR or AODV-PA that maintain the complete route to destinations.
Dynamic Source Routing (DSR) is a source routing protocol, and it maintains a route cache that contains multiple, complete source routes to destinations. Therefore, instead of choosing polar nodes randomly, the source node constructs the set of all nodes along routes that pass through the destination or terminate at the destination. This set is called the *Polar Node Set*.
Figure 4.10 shows the path traversed by packets corresponding to the poles chosen. If node S wishes to transmit packets to node D, and at S the routes that pass through or terminate at D are $S \rightarrow L \rightarrow M \rightarrow N \rightarrow P \rightarrow D$ and $S \rightarrow L \rightarrow M \rightarrow Q \rightarrow R \rightarrow D \rightarrow U \rightarrow T$, then the *Polar Node Set* corresponding to D would be $L, M, N, P, Q, D, R, U, T$. Hence, poles would be chosen among these nodes. Choosing M and P as the poles (Case (a)) will not cause an increase in the path length, while choosing Q and U (Case (b)) as the poles will result in a slight increase in path length. Notice that not choosing poles in the direction of the destination will always result in a slight increase in the path length. For example, in Figure 4.10(a), choosing pole P followed by M for a packet, would result in a greater path length than choosing M followed by P.
The advantages of employing source-route based routing protocols such as DSR are:

- The number of hops to reach the destination is close to optimum, since poles be chosen along the path to the destination.

- Since the poles are chosen from DSR's route cache, it is very likely that routes to these nodes are already known. Thus, the number of new route discoveries to the poles is minimal.

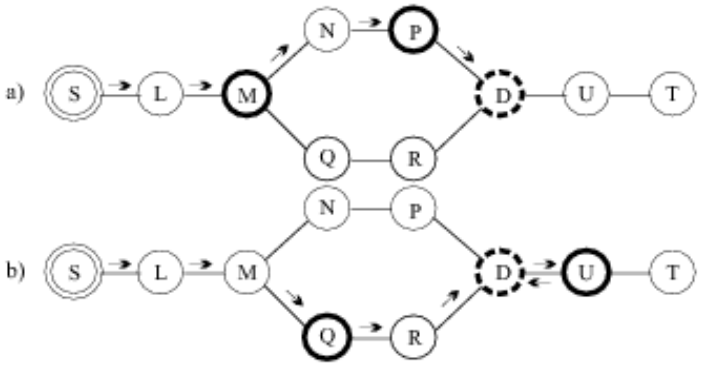

Figure 4.10: Examples of sets of poles and the corresponding path of packets, where node S is the source and node D the destination. The arrows denote the path of the packets.

4.3.3.2 Caching and Pre-determining Poles

For a source and destination pair, the *Polar Node Set* remains fairly constant over short time intervals. Also, communication between nodes is usually bursty or continuous, rather than sporadic. Therefore, it is advantageous to cache the *Polar Node Set* and update it at regular intervals, rather than reconstruct it from the route cache poles for every packet transmitted.
A further improvement to this strategy is to cache a subset of possible poles for each destination in a *Pre-selected Polar Node Table (PPNT)*, as oppposed to caching the entire *Polar Node Set*. The *Polar Node Set* to each destination is constructed at the beginning of each interval. The entries of the PPNT are then populated with 0,1 or 2 randomly chosen poles picked from the *Polar Node Set*. The PPNT is constructed only once at the beginning of each interval. hence, instead of choosing the polar nodes from the *Polar Node Set* for each packet, the source node randomly picks an entry from the PPNT that contains the poles to be used for the packet. An upper bound is imposed on the number of entries in the PPNT to limit its size. The number of different routes that can be chosen for a destination is limited by this bound. Table 4.1 shows a sample PPNT for the network shown in Figure 4.10.
The main advantage of using the PPNT is that all packets to a destination are sent through only a few routes dictated by the entries of the PPNT. This results in fewer route discoveries, and consequently lower control overhead and packet delivery time. Also, less time is spent on deciding the number of poles and choosing polar nodes. Thus utilizing the PPNT further reduces the packet

Num. of Poles	Pole 1	Pole 2
1	M	**N/A**
2	M	P
1	T	**N/A**
2	P	M
2	R	P
0	**N/A**	**N/A**

Table 4.1: Example Pre-selected Polar Node Table (PPNT) for node D at node S

delivery time. In fact, if only the header of the packet is encrypted as opposed to the entire packet, the encrypted header can be pre-calculated and stored in the PPNT. This will result in significantly lower delay and processing overhead, since the process of encrypting the packet header with the public key of each pole is performed just once at the beginning of each interval, instead of for every packet transmitted.
AD-MIX periodically refreshes the contents of the PPNT to accommodate changes in the route cache due to dynamic changes in network topology. in order to accommodate dynamic changes in network topology, it is beneficial to vary the refresh rate with the mobility of the node and the number of RERRs received.

4.3.3.3 Forcing loopback

A selfish node is forced to forward packets that are not immediately destined for itself because there is a possibility that it is the ultimate destination of these packets; i.e., these packets can loop back to the node. If the possibility of packets looping back is very small, then a selfish node may be tempted to risk dropping packets not destined for it. To prevent this, AD-MIX can force a specified percentage of packets that loop back. Increasing the percentage of packets that loop back increases the probability of a selfish node dropping its own packets. This results in increased cooperation in forwarding packets destined for other nodes.
AD-MIX forces loopback by either choosing a node beyond the destination, with the destination node along its path as a pole; or, if no such node is known,

choosing the destination as the first pole and any node along the path as the second pole.
Figure 4.11(a) illustrates how loopback is enforced when a node beyond the destination is known to the source. If node S wants to communicate with D, and it has a path to T containing D along it, then it can force loopback by choosing T as the pole. If D is selfish and drops the packet after noticing that the destination is T, it will drop its own packet. However, if S does not know

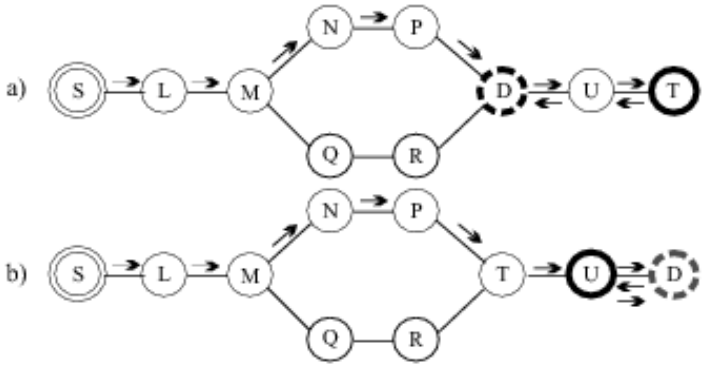

Figure 4.11: Two cases of forced loopback

any node beyond D, then it would not be able to employ the previous scheme. In such a case, the source can send looped back packets with two poles. The first pole is the destination itself, while the second pole could be any node along the path. This is depicted in Figure 4.11(b), where U is chosen as the second pole. here again, if the node behaves selfishly, it will drop packets destined for itself.

4.3.4 Encouraging Participation using AD-MIX

In the previous sections we have presented the operation of the AD-MIX protocol. To show that AD-MIX encourages participation, it is sufficient to show that it is impossible for any node to deduce the ultimate destination of any packet passing through it. In their paper they try to prove that a minimum of two poles is necessary to achieve this:

- Zero poles:
  If the packets are not encrypted, or encrypted with just the public key

of the destination, then a selfish node can determine whether it is the intended destination of the packet by looking at the destination address. It can drop the packet if it is not the destination.

- One pole:
  If every packet is encrypted with the public key of one pole, then the non-polar nodes would have no way of determining the true destination of the packet, since the destination specified in the packet header may either be the next pole or the destination. Since it is possible that the non-polar node itself could be the destination, a selfish non-polar node is forced to forward packets not destined for itself, or risk dropping its own packets. However a polar node, after decrypting the packet, can determine that the packet is not destined for it. Since only one polar node is used, the node to which the polar node would forward the decrypted packet has to be the packet's ultimate destination. Hence, a selfish polar node could drop the packet without the risk of losing messages sent to it.

- Two or more poles:
  When two or more poles are used, it becomes impossible for event the polar nodes to determine the ultimate destination of the packets they receive. By padding the header with variable number of times a packet is encrypted, since this information cannot be obtained from either the size or the destination of the packet. Therefore, a polar node cannot be sure, even after decrypting the packet, whether the next address of the packet is the destination, unless it has information to suggest that it is the second pole of the packet. Since no node possesses this information, the destination of packets are successfully obscured.

Hence, in order to conceal the destination of the packet from both polar and non-polar nodes, at least two poles must be chosen.
Nonetheless, using two or more poles for each packet will incur a considerable control overhead and delay in mobile ad hoc networks. AD-MIX overcomes this by choosing variable number of poles (between 0 and 2) for each packet. Hence, even if no poles are chosen or if only one pole is chosen for an particular packet, the intermediate nodes will be forced to propagate the packet since they cannot deduce the number of poles chosen for the packet. A node cannot route out the possibility of the packet looping back to it.
If both the poles chosen for a packet are identical, AD-MIX's assumption that the intermediate nodes do not have sufficient information to deduce the packet's ultimate destination is broken. If a pole decrypts a packet twice, it will know that the node it should forward the packet to is the packet's ultimate destination. A selfish pole can therefore drop such packets. To prevent this, a route should never use two identical poles.

4.3.5 Security and Other Considerations

In this section we present attributes of AD-MIX that enable it to achieve more than encouragement of participation. We also present circumstances under which AD-MIX may not be able to achieve its objectives effectively.

4.3.5.1 Secure Communication

Since the source encrypts the contents of a packet up to three times before transmitting it, at no instant along the route are the contents of the packets available unencrypted. Only the final destination can decrypt the packet to view its contents. The protocol design therefore provides information security as well as prevents *man-in-the-middle* and *replay attacks*. Hence AD-MIX provides a secure path for communication of messages from the source to the destination.

4.3.5.2 Desire to Communicate

AD-MIX assumes that all nodes in the network want to participate in network communications. This is a reasonable assumption since the purpose of a node joining an ad hoc network is to communicate with other nodes in the network. However, if a particular node does not expect to receive any packet, then it can drop all packets passing through it. Such nodes may be classified as malicious.

4.3.5.3 Colluding Nodes

If two nodes collude by agreeing not to use any poles to transmit data between themselves, then the destination node can safely drop all packets not destined to it, since it is sure that the source does not use any poles while transmitting to it. Under such circumstances, the current design of AD-MIX will not work. However, such colluding nodes can be handled in two ways:

- Use other mechanisms to detect nodes that do not forward packets on behalf of other nodes in the network. The action taken against colluding nodes may vary from merely reporting misbehavior to isolating them from the network. The action taken should deter other nodes from colluding.
- Assume the presence of a "tamper-proof" layer at each source node that cannot be by-passed. All packets transmitted by the node are forced

to pass through this layer. This layer decides the set of poles for each transmitted packet. This prevents collusion since the choice of polar nodes is no longer under the control of the user of the source node.

4.3.5.4 Discarding Control Packets

A node may decide to drop all control packets passing through it except route request packets destined for itself. If this happens, no node in the network will have any routes through this node; any route with the node's address will be a route route terminating at the node. This implies that if AD-MIX is not used, then by dropping all control packets through the node, it can ensure that no packet will ever be sent to it unless the packet is destined for it. A selfish node can thereby save its resources. However, employing AD-MIX mitigates this behavior since if the source node possesses a path to the selfish node, it can use the selfish node as a pole to transmit packets to other nodes along the path.

4.3.5.5 Traffic Analysis and Anonymization

Since AD-MIX employs a model similar to the mix servers, packets traveling between a source and a destination do not follow the same path. Even if they follow the same path, choosing different poles will make the packets appear dissimilar, making it extremely difficult to gather information about on-going sessions by traffic analysis. hence, AD-MIX also provides an effective mechanism to counter attacks against privacy. Also, since the destination of the packet is hidden from all nodes in the network, AD-MIX inherently promotes anonymization.

4.3.6 Summary

In the paper of AD-MIX, they addressed the issue of selfishness of nodes in mobile ad hoc networks. They proposed a protocol to encourage participation. By choosing polar nodes appropriately overhead caused by forced loopback can be reduced.

Chapter 5

Privacy in Mobile Ad Hoc Networks

In this chapter we will give a definition of the privacy requirements of our protocol *MixRoute* based on Federrath et al. [10].

5.1 Requirements resulting from Data Protection

For mobile communication systems intended for broad use, the following requirements resulting from data protection should be met:

- **Protection of Confidentiality**
  1. *Message contents* should be kept confidential towards all parties except the communication partners.
  2. *Sender* and/or *receiver* of messages should stay anonymous to each other, and third parties should be unable to observe their communication.
  3. Neither potential communication partners nor third parties should be able to locate other nodes.

- **Protection of Integrity**
  1. Forging *message contents* (including sender's address) should be detected.
  2. The recipient of a message Y should be able to prove to third parties that entity X has sent message Y.
  3. The sender of a message should be able to prove the sending of a message with correct contents, if possible, even that the receiver received the message.
- **Protection of Availability**
  1. The communication network enables communication between all parties who wish to communicate.

Confidentiality requirements must be enforced by the prevention of the gathering of personal data. There is no other way known to achieve privacy.
In the following, we will show how by technical means for data protection we can provide security for the nodes in the network.

5.2 Realization of Data Protection Requirements

How and where is it possible to realize data protection requirements? The fact of mobility makes it difficult to apply well known concepts in the same way as in fixed networks. After outlining these basic concepts some ideas are given which could point into the right direction.

5.2.1 Basic Concepts

Security problems may be solved by using methods such as end-to-end encryption and link-to-link encryption. The anonymity of participants can be protected by using certain mix nodes.

5.2.2 Protection of User Data

Requirement of confidentiality of *message content* means, trusted communication between two participants of the same and of other networks must be

possible. The same must be true for the integrity requirements. These can be achieved by encryption, digital signatures and authentication codes. In fact, confidentiality of *message content* can be accomplished by end-to-end encryption. Forging *message content* should be detected for example by, a hash-value of a message is digitally signed. For a sender to prove that it is the actual sender of a message a digital signature of the sender of the message is necessary. Fulfillment of the requirement needs a signed receipt from the recipient.
Cryptography is only applicable if the following conditions are true:

- The different services and different network systems need to match corresponding encryption methods and protocols. But this seems to be more a legal (political) and economical than a technical problem.
- User channels must be bit-transparent, i.e. bit to be transmitted on the signal path must not be changed or interfered with. The minor change of bits could mean a loss of integrity on the signal path. Furthermore, a change of only one bit would be followed by n increased rate of errors since encryption systems usually produce a strong dependency between bits.
- Even bit-transparency is not implemented in every already realized and standardized network. Considering these aspects the integration of networks and services must be planned carefully.

5.2.3 Protection of Data

The following concepts show the possibility of developing networks which fulfill our data protection requirements.

5.2.3.1 Link-to-link Encryption

The contents of a message can be sufficiently hidden by end-to-end encryption at the ISO layer 4. If protocols of the layers 1 to 3 also contain personal data then it is also necessary to protect this information by link-to-link encryption. This information could be the address of a node.

5.2.3.2 Recipient Anonymity by Broadcast Addressing Attributes

Receiving a message can be made completely anonymous to the network by delivering the message (possibly end-to-end encrypted) to all nodes. If the

message has an intended recipient, it has to contain an attribute by which he and nobody else can recognize it as addressed to him. This attribute is called an *implicit address*. it is meaningless and only understandable by the recipient who can determine whether he is the intended node. in contrast, an *explicit address* describes either a place in the network.
Implicit addresses can be distinguished according to their visibility, i.e. whether they can be tested for equality or not. An implicit address is called *invisible*, if it is only visible to its recipient and is called *visible* otherwise.
Invisible implicit addresses, unfortunately very costly, can be realized with a public key cryptosystem. *Visible implicit addresses* can be realized much easier: Users choose arbitrary names for themselves, which can then be prefixed to messages.
Another criterion to distinguish implicit addresses is their distribution. An implicit address is called *public*, if it is known to every node and *private* if the sender received it secretly from the recipient as a return address or by a generating algorithm the sender and the recipient agreed upon.
Public addresses should not be realized by visible implicit addresses to avoid the linkability of the visible public address of a message and the addressed user.
Private addresses can be realized by visible addresses but then each of them should be used only once.

5.2.3.3 Sender Anonymity by using mix-net

Unlinkability of sender and recipient can be realized by a special node called a mix, which collects a number of messages of equal length from many distinct senders, discards repeats, changes their encodings, and forwards the messages to the recipient of a message from everybody but the mix and the sender of the message. Change of encoding of a message can be implemented using a public-key cryptosystem. Since decryption is a deterministic operation, repeats of messages have to be discarded. Otherwise, the change of encoding does not prevent tracing messages through mixes: Simply count the number of copies of each message before and after the mix.
By using more than one mix to forward a message from the sender to the recipient, the relation is hidden from all attackers in the network who do not control all mixes which the message passed, nor have the cooperation of the sender. Mixes should be independently designed and produced and should have independent operators, otherwise a single party is able to control a communication.
One method for achieving sender anonymity i.e. of making unclear when a message was sent, is *dummy-traffic*. That means each node sends among the real message a number of meaningless messages.
Unfortunately the above described concepts are hardly feasible for the protection of the radio interface, e.g. dummy-traffic is not acceptable due to limited

energy capacity and bandwidth.

5.3 Summary

This chapter described the privacy requirements for our protocol. In the next chapter we will present the design of our protocol based on the requirements in this chapter.

Chapter 6

Design

Consider a wireless ad hoc network in which a subset of mobile nodes are mixes. Mixes cooperate to provide anonymous connection service to any source/destination pairs, regardless of node type. In other words, anonymous connections can be established between two non-mix nodes, one non-mix node and one mix, and two mixes. For the ease of presentation, an assumption is made: The source and destination of an anonymous connection are both non-mix nodes. The original mix-net is based on public key cryptosystem. Assuming that each mix i generates a pair of keys K_i and K_i^{-1}, the public key K_i is made known to all users and the private key K_i^{-1} is never divulged. Chaum described a way of delivering message without disclosing sender/recipient relationship, as follows [6]. First, the sender S decides a *mix route*, which is a sequence of mixes. Second, S "seals" a message M for delivery by successively encrypting M with public keys of the mixes in the route. Say the mix route is $M_1 \rightarrow M_2 \rightarrow ... \rightarrow M_k$, and the encryption of a message M with key K_X is denoted $K_X(M)$. The sealed message M_S would be of the form

$$M_S = K_1(R_1, K_2(R_2, ..., K_k(R_k, M)...))$$

where R_i is a random string attached to a message before each encryption. Only the holder of the private key K_i^{-1} can interpret a message encrypted with the public key K_i. So the sealed message will be sent to M_1, who can remove one layer of encryption, throw away R_1, and send the remainder of the message to the next mix M_2. Each mix in the route follows the same procedure, and the

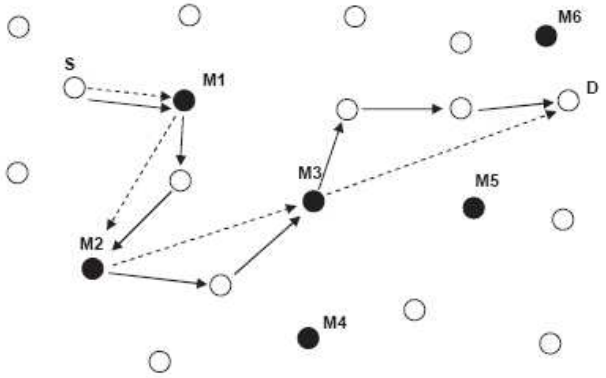

Figure 6.1: A mix-net example in a wireless ad hoc network

last mix M_k will finally deliver M to its recipient node. M can be encrypted with the recipient's key or plain text. Note that each mix knows only the previous and next mix, except that the first and last mix know the sender and recipient of the message respectively. Hence, unless all mixes are compromised, an adversary cannot determine both sender and recipient of the message.
The purpose of a mix is to hide the correspondences between the messages in its input and those in its output. How well a mix achieves this goal depends on a number of factors, such as the adversary's ability, the mix flushing algorithm, the mix input size (i.e., traffic load), etc. Assuming the same adversary and attack model as in ANODR [18], i.e., a powerful adversary with unbounded eavesdropping capability but bounded computing and node intrusion capability, this means that (i) the adversary can eavesdrop transmissions on all wireless links but cannot break public key or symmetric key crypto-systems to discover the contents of the messages without acquiring the corresponding keys; (ii) the adversary may capture and compromise mixes but cannot successfully compromise more than K members during a time window T. In addition, intrusion detection is not perfect. So a compromised mix exhibiting no malicious behavior will stay in the network and participate in relaying traffic. This means that the untraceability of an end-to-end connection can never be guaranteed.
An example of ad hoc mix-net is given in Figure 6.1, where mixes are indicated by dark nodes. In this setting, each packet from node S to node D pass through three mixes, M_1, M_2 and M_3. Hence the mix route created for the source-destination pair (S, D) is $M_1 \rightarrow M_2 \rightarrow M_3$, as shown by the dashed-line. The solid line draws the physical route that data packets actually take. Clearly, a mix route is a logical route, not a physical route, and the mix-net is an overlay

network.

6.1 Design of MixRoute

In this section, our enhanced mix route algorithm is presented which is called *MixRoute*. The purpose of MixRoute is to find mix routes for an end-to-end connection. Several design goals are set for the algorithm. First, connection anonymity should not be violated during the mix route discovery process. Second, the algorithm should find a short mix route based on the current network topology. As the network topology changes, the algorithm should update the mix route. Third, the algorithm should have low and bounded overhead.
First the algorithm is described, followed by a detailed discussion. MixRoute consists of two independent processes: mix advertisement (using MADV messages), and mix route discovery and update (using DREG, RREQ and RUPD messages). It should be emphasized that the "mix route discovery" process runs on top of any underlying routing protocol. In essence, the mix route discovery process finds routes consisting of "virtual links" between mix nodes - a virtual link in the mix-net is a path in the physical network.

- The purpose of mix advertisements is for the mix nodes to announce their presence to non-mix nodes. Each non-mix node tries to pick the closest mix node as its first mix node on the route - the closest mix node serves a function in anonymous routing as seen below.

- Due to node mobility, each non-mix node may dynamically change the mix node chosen as its nearest mix node. To make each mix node aware of its nearest mix node relationship with non-mix nodes, the non-mix nodes use DREG messages to register at their nearest mix nodes.

- In this approach, when a node S needs to find an anonymous route (through one or more mix nodes), it sends a RREQ message to the destination D via a custom mix route formed by a set of randomly chosen mixes or by S's closest mix node. The custom mix route may not be the right choice from performance perspective, therefore, the rest of the mix route discovery process attempts to find a better mix route for the connection. For instance, if S chooses a mix M_3 randomly, then the mix route for the RREQ will be $S \rightarrow M_3 \rightarrow D$. The RREQ packet is routed from S to M_3 using the underlying routing protocols (we have chosen DSR [16]), and from M_3 to D similarly. When D receives the RREQ, the destination node realizes that it is an endpoint for an active connection. Therefore, it registers with its closest mix node by sending a DREG message.

- Any mix node that has a non-empty list of registered non-mix nodes periodically transmits a RUPD message as elaborated later. The purpose of RUPD transmissions is to allow a source node to discover a mix route regarding a particular destination node (A RUPD message contains a list of all destination nodes currently registered at the mix node who creates the message).

In the rest of this section there will be further elaborations on the above algorithm.

6.1.1 Mix advertisement

In the following is introduced a low-cost mix advertisement algorithm for non-mix nodes to find the closest mix:

1. Every mix periodically broadcasts mix advertisement (MADV) messages to announce its presence to non-mix nodes in the neighborhood. The time interval between two consecutive advertisements is ADVERTISE_INTERVAL. MADV from mix M has message format:

   $$\langle MADV, M \rightarrow ALL, seqnum, radius \rangle$$

   where (i) seqnum together with M's address uniquely identify a MADV message. (ii) The *radius* value indicates how far the message has propagated. When the message is created, it is set to 0.

2. A non-mix node learns mixes in its neighborhood from received MADV messages and maintains the closest mix information, which is also the node's closest mix. As time passes, the node's neighborhood may change. Therefore, a non-mix node's closest mix is not constant. It is also possible that a non-mix node loses connectivity with its current closest mix. So if a non-mix node does not receive MADV packets from the current closest mix for a time interval of length 2 * ADVERTISEMENT_INTERVAL it switches to a new closest mix. A non-mix node only retransmits MADV messages from its closest mix. Every time when a MADV message is retransmitted, the *radius* value in it is incremented by 1.

3. A mix node discards MADV messages that it receives.

The described algorithm is unlike the conventional, network-wide flooding algorithm. Each MADV message has a limited flooded area. Typically, it only

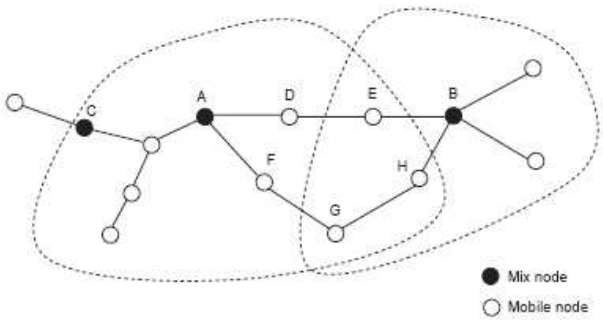

Figure 6.2: Flooded area of mix advertisements

arrives at nodes that are closer to it than to any other mixes.
An example is used to illustrate this idea. In Figure 6.2, the border of two mixes' flooded area is shown by dashed line. A is the closest mix to D. So D will retransmit A's MADV messages. But E does not retransmit A's MADV's it received from D because mix B is closer to it than A is. The validity of this algorithm can be shown by considering a non-mix node that receives two MADV messages, one from the closest mix M, another from a farther mix X. The *radius* values in the two messages must satisfy $radius(M) < radius(X)$. Suppose that the node retransmits both messages: A neighboring node that receives the two messages will find that the above relationship still holds because the *radius* values in both messages are increased by 1, respectively. In other words, based on these two messages, X can never be closer to any downstream nodes than M is. So it is unnecessary to forward the MADV messages from X.

6.1.2 Mix Route Discovery and Update

The mix route discovery process might be best described by an example. Figure 6.3 shows a mix-net of 6 mixes (marked as dark).
Node S wishes to find a mix route for an anonymous connection destined to node D. The mix route discovery process can be divided into three phases:

1. *RREQ phase*: S assembles a RREQ message and sends it to D via a custom mix route. As we mentioned, a custom mix route can be a random route consisting of randomly chosen mixes, or be the closest mix of S as in this example. The RREQ message is a unicast message. So S

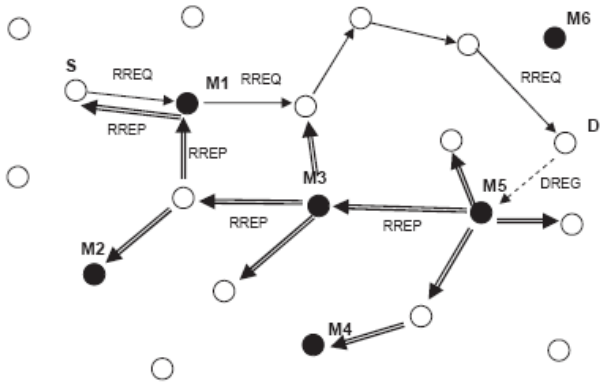

Figure 6.3: Mix Route Discovery Process

can encrypt the content of the message with D's public key to prevent tracing of the message by an attacker. The RREQ packet may be lost during transmission. So a timeout-based retransmission mechanism must be activated by S.

2. *DREG phase*: When D receives a RREQ message, it knows that it is destination of a new end-to-end connection. If D did not yet register at its closest mix (M_5 in this example), it does so by sending a Destination Registration (DREG) message to the mix. Let M be D's closest mix. The DREG message would have format

$$\langle DREG, D \rightarrow M, seqnum \rangle$$

D must send DREG messages periodically to maintain its association with the mix. There are several reasons for this design. First, DREG messages may be lost during transmission and never reaches the mix. Second, as network topology changes, D may switch to a different closest mix. In this case, D simply sends DREG messages to the new closest mix and increases the *seqnum* in it. The old closest mix may learn this change from one of two events. One is expiration of D's registration record because there is no new DREG message arriving from D. Let DREG_INTERVAL be the time interval between two consecutive DREG messages. The expiration time of a destination node's registration at mix is set as 2 * DREG_INTERVAL in the algorithm. Another is receiving RUPD messages from D's new closest mix (explained below).

3. *RUPD phase*: Every mix maintains a list of registered destination nodes. If the list is not empty, it periodically broadcasts RUPD messages. The

time interval between two consecutive broadcasts is RUPD-_INTERVAL. RUPD message from a mix M is of the format

$$\langle RUPD, M \rightarrow ALL, seqnum, l, path \rangle$$

where (i) *seqnum* together with M's address uniquely identify a RUPD message. (ii) l is the list of destination nodes currently registered at M. Each entry of the list includes node address and the latest DREG *seqnum*. (iii) *path* records a mix route that the packet has traversed during flooding. Initially, *path* contains M, the initiator of the message.
The flooding of a RUPD message proceeds as follows. The initiator mix broadcasts the message locally. If a node X that receives the RUPD message has pending data packets in its queue addressed to destination node(s) in l, then it copies the mix route in *path* and uses the reverse mix route in delivering those data packets[1]. If X is a mix, then it checks whether any destination node in l carries a higher DREG *seqnum* and updates its own list. When the above processing is completed, X retransmits the RUPD message, and if X is a mix, it appends its ID to the *path* before retransmitting. It is possible that X receives the same RUPD message for multiple times. To ensure that a RUPD message is retransmitted only once, X keeps a record of each RUPD message it retransmitted. However, from the multiple RUPD messages that arrive via different paths, X may obtain multiple distinct mix routes to the same destination node. In Figure 6.3, the retransmissions of RUPD message are indicated by double arrows. It is shown that S will find a mix route $M_1 \rightarrow M_3 \rightarrow M_5$ for its connection to D.

From the above description, we know that the RUPD message is flooded along the shortest path tree rooted at the initiator mix. For the same destination node, different source nodes receive different mix routes and the minimum length of each mix is 1. The idea is that each mix caches the mix routes it received and broadcasts them along with MADV messages. The source node of a connection will use the mix Route received from its closest mix node, which contains at least two mixes.
The update of mix route for an anonymous connection is realized by periodically RUPD broadcasts. If a node is not destination of any active connection, it should stop sending DREG messages to its closest mix node. It is assumed that an in-band protocol exists for the source node to inform the destination node of connection termination.

[1] If the RUPD packet is received from an unidirectional link, it should be discarded because there is no reverse link

6.2 Security Analysis

In this section, an analysis of the security aspects of MixRoute is provided. Raymond [29] presents a good survey of known attacks against mix-nets. So the focus will be on new attacks that employ vulnerabilities in the mix route algorithm to reveal source and destination of an anonymous connection.
During the mix route discovery process, an attacker may employ the correlation between RREQ message and DREG message to reach its goal. For instance, if the attacker observes that node S sends a RREQ message and node D sends a DREG message shortly after, then it is very likely that S and D are two end-points of a new anonymous connection. We have mentioned that RREQ messages can be encrypted with destination node's key so that only the destination node can interpret the contents of the messages. However, message encryption is not effective when there is no sufficient cover traffic. In this case, S can send multiple dummy messages, i.e. to itself, before it sends real RREQ messages.
During the mix route update process, a long-lived connection is subject to intersection attack due to change of mix route. In a high-mobility network, it is very likely that the source node of a connection receives multiple updates of mix route during the connection lifetime. Assuming that the source node uses the shortest mix route all the time, the attacker can perform attack as follows. The attacker finds the shortest mix routes and the first mixes in the route to a destination node changes and the new mix route has a different "first mix" than the old mix route had, the attacker can observe whether the source node "shifts" data traffic from the connection to the old "first mix" onto the connection to the new "first mix". The attacker has a better chance to succeed when the source node has only a few connections. To prevent this attack, a perfect, but very costly, solution is that the source node maintains constant traffic loads to each mix by use of dummy traffic. A less costly solution is that the source node splits the data traffic between multiple mix routes, which are learned from RUPD messages. This should complicate traffic analysis and reduce dummy traffic load as well. However, either solution decreases network performance.

6.3 Cost Analysis

In this section, an analyze of the control overhead of the proposed algorithm is provided. The total number of control packets generated during a time window T is counted. For broadcast control packet, retransmissions of the packet are counted individually.
Let n be the total number network nodes, and m be the number of mixes. For

analysis purpose, it is assumed that c end-to-end connections (each with different destination node) are set up during the window T. In the algorithm, MADV packets are generated and flooded by each mix at an interval of ADVERTISE_INTERVAL. During each advertisement cycle, every non-mix node retransmits MADV packet (from the closest mix node) only once. So the total number of transmissions of MADV packets by all nodes is n. RREQ packets are generated by the source node of each end-to-end connection. Assuming that all RREQ packets are successfully delivered, the total number of RREQ packets during the time window T must be c. The destination node of each

Packet Type	Packet Count	Asymptotic Upper Bound
MADV	$n\frac{T}{ADVERTISEMENT\_INTERVAL}$	$O(n)$
RREQ	c	$O(c)$
DREG	$c\frac{T}{DREG\_INTERVAL}$	$O(c)$
RUPD	$m\frac{T}{RUPD\_INTERVAL}n$	$O(mn)$

Table 6.1: Analysis of Control Packet Load

end-to-end connection generates DREG packets periodically at an interval of DREG_INTERVAL. So the total number of DREG packets during the time window T must be $c\frac{T}{DREG\_INTERVAL}$. In the worst case, all mixes need to generate RUPD packets. Each RUPD packet is flooded to all nodes and each node retransmits each RUPD packet only once. Hence, the total number of transmissions of each RUPD packet amounts to n.
The above analysis is summarized in Table 6.1. It is shown that majority of the control overhead is due to the periodical flooding of RUPD packets. In the worst case, the overall control packet load is $O(mn + c)$. But on the average, the number of mixes that broadcast RUPD packets is expected to be less than m.

Chapter 7

Network Simulator version 2

The Network Simulator version 2 (ns-2) is a deterministic discrete event network simulator, initiated at the Lawrence Berkeley National Laboratory (LBNL) through the DARPA funded Virtual InterNetwork Testbed (VINT) project. The VINT project is a collaboration between the Information Sciences Institute (ISI) at the University of Southern California (USC), Xerox's Palo Alto Research Center (Xerox PARC), University of California at Berkeley (UCB) and LBNL.

7.1 ns-2

ns-2 was initially created in 1989 to be an alternative to the REAL Network Simulator. Since then the uses and width of the ns project has grown significantly. Although there are several different network simulators available today, ns-2 is one of the most common. ns-2 differs from most of the others by being an open source software, supplying the source code for free to anyone that wants it. Whereas most commercial network simulators will offer support and a guarantee but keeping the money making source code for themselves. The release of the source code helps users to create their own functions an subprograms, but also makes it easier to implement them into the ns-2 environment. One of the main benefits for the ns project group releasing the source code is that independent researchers can help in the development of ns-2. It is fairly common

that a researcher contributes with the code of an non-implemented protocol or algorithm, after constructing it for his own studies.
It is worth noting that ns-2 is a research effort and not a commercial software release. The difference is that there are very few people in the ns project group compared to an ordinary software, leading to difficulties in supporting all the users. That problem has lead to the solution of having a huge mailing list (`http://mailman.isi.edu/mailman/listinfo/ns-users`) for anyone interested, as well as a complete archive of all the mails ever been sent to this mailing list. The mailing-list is based on the idea of user helping user, taking the load of the ns project group. The mailing-list and the archives are a huge help for all users of ns-2, no matter old or new, since usually someone else has had the same problem before.
Another important thing to to remember is that ns-2 is an ongoing project and hence not completed product. This being the reason why it is free and offers no support except the mailing-lists. The people that are in charge of the project heavily relies on the users to find bugs and faults and reporting these when discovered. This also leaves the validating of results to the user, but the user is not alone so help is just an email away.
The most common protocols are so well used and checked so the main worries are the new implementations. New implementations usually start out as a research assignment not linked to the ns project group. Since the project group does not have a full company helping them in verification and implementation they have no possibility to do everything themselves thus encouraging any help they can get.

7.2 The ns-2 structure

ns-2 is made up of hundreds of smaller programs, separated to help the user sort through and find what he or she is looking for. Every separate protocol, as well as variations of the same, sometimes have separate files. Though some are simple spinoffs, still dependent on the parental class. Considering the sheer size of ns-2 it would be all but impossible to gain an understanding of the code if it was just a few gigantic code files.

7.2.1 C++

C++ is the predominant programming language in ns-2. It is the language used for all the small programs that make up the ns-2 hierarchy. C++, being one of the most common programming languages and specially designed for object-

oriented coding, was therefore a logical choice what language to be used. This helps when the user wants to either understand the code or do some alterations to the code. There are several books about C++ and hundreds, if not thousands, of pages on the Internet about C++ simplifying the search for help or answers concerning the ns-2 code.

7.2.2 OTcl

Object Tcl (OTcl) is object-oriented version of the command and syntax driven programming language Tool Command Language (Tcl). This is the second of the two programming languages that ns-2 uses. OTcl is used as an front-end interpreter in ns-2. Linking the script type language of Tcl to the C++ backbone of ns-2. Together these two different languages creates a script controlled C++ environment. This helps when creating a simulation, simply writing a script that will be carried out when running the simulation. These scripts will be the recipe for a simulation and is needed to set the specifications of the simulation itself. Without a script properly defining a network topology as well as the data-flows, both type and location, nothing will happen. For a more in depth presentation of these scripts have a closer look at the introduction and related chapters in the ns-2 manual.

7.3 Nodes

A node is exactly what is sounds like, a node in the network. A node can be either an end connection or an intermediate point in the network. All agents and links must be connected to a node to work. There are also different kinds of nodes based on the kind of network that is to be simulated. The main types are node and mobile node, where node is used in most wired networks and the mobile node for wireless networks. There are several different commands for setting the node protocols to be used, for instance what kind of routing is to be used or if there is a desire to specify a route that differs from the shortest one. Most commands for node and mobile node can be found in the ns manual. Nodes and the closely connected link creating commands, like simplex_link and duplex_link, could be considered to simulate the behavior of both the Link Layer.

7.4 Agents

Agents is the collective name for most of the protocols you can find in the transport layer. In the ns-2 manual they are defined as the endpoints where packets are created and consumed. The agents in ns-2 are all connected to their parent class, simply named `Agent`. This is where their general behavior is set and the offspring classes are mostly based on some alterations to the inherent functions in the parent class. The modified functions will overwrite the old and thereby change the performance in order to simulate the wanted protocol. Taking the TCP Tahoe agent, which is parent of all TCP implementations, as an example, it inherits its base functions from the parent class `Agent` just adding the necessary functions needed to gain TCP behavior. The TCP Reno agent then inherits the behavior of TCP Tahoe, just substituting the functions that needs to be modified to simulate the TCP Reno behavior instead of writing a whole new program. Since ns-2 is based on the TCP/IP suite most of the implemented agents are the protocols that can be found in the TCP/IP Transport layer. There is, for example, the UDP protocol, the RTP protocol and several TCP clones.

7.5 Applications

The applications in ns-2 are related to the Application Layer in the TCP/IP suite. The hierarchy here works in the same fashion as the in the agents case. The ns-2 applications are used to simulate some of the most important higher functions in network communication. Since the purpose of ns-2 is not to simulate software, the applications only represent some different aspects of the higher functions. Only a few of the higher layer protocols has been implemented, since some are quite similar when it comes to using the lower functions of the TCP/IP stack. For instance there is no use adding both a SMTP and a HTTP application since they both use TCP to transfer small amounts of data in a similar way. The only applications incorporated in the release version of ns-2 is a a number of different traffic generators for use with UDP and telnet and FTP for using TCP. All the applications are script controlled and when concerning the traffic generators, you set the interval and packet-size of the traffic. FTP can be requested to send a data packet whenever the user wants to, or to start a transfer of a file of an arbitrary size. If starting an FTP transmission and not setting a file-size the transmission will go on until someone calls a stop.

7.6 NAM

The Network Animator NAM is a graphic tool to use with ns-2. It requires a nam-trace file recorded during the simulation and will then show a visual representation of the simulation. This will give the user the possibility to view the traffic packet by packet as they move along the different links in the network. NAM offers the possibility of tracing a single packet during its travel and the possibility to move the nodes around for a user to draw up his network topology according to his own wishes. Although since the simulation has already been performed there is no possibility for the user to change the links or any other aspect of the simulation except the representation. NAM is dependant on the existence of an X-server in order to be able to open a graphical window. Therefore there has to be a version of X running if NAM is to work.

Chapter 8

Implementation

In this chapter we will describe the implementation of *MixRoute*. The protocol is implemented partly in C++ and partly in OTcl. The protocol consists of several files and some necessary changes in the ns-2 source files.
The C++ files for MixRoute are placed in a subfolder to ns-2 called `/mix`. This folder contains the following files:

- `mixagent.h`
- `mixagent.cc`
- `mix_hdr.h`
- `mix_hdr.cc`
- `mix_routecache.h`
- `mix_routecache.cc`

The OTcl files for MixRoute including the test script are placed in the subfolder ns-2/tcl/anonymity-test. The folder contains the following files:

- `mixroute.tcl`

- `mix.tcl`
- `mixroute.tr`
- `out.nam`
- `cbr-n50-mc10-r4`
- `scen-1000x1000-mix5`
- `scen-1000x1000-n50`

In the following sections the implementation of *MixRoute* will be described.

8.1 Mix agent

Agents represent the endpoint where the network-layer packets are constructed or consumed, and are used in the implementation of the protocol at various layers. The class `MixAgent` is the implementation of the agent, which is partly implemented in C++ and partly in OTcl. The implementation of the agent works as an overlay protocol at the application level. The mix agent uses the DSR agent `DSRAgent` as an underlying routing protocol. Most of the methods of the DSRAgent are integrated in `MixAgent` slightly modified to be used to handle packets in `MixAgent`. The files for the DSR implementation can be found in the subfolder `/dsr` to ns-2. In the

8.2 Mix packet header

We have defined a specific header type to be used in packets. The structure `hdr_mix` defines the layout of the mix packet header. This structure definition is only used by the compiler to compute byte offsets of fields; no objects of this structure type are ever directly allocated. The structure also provides member functions which in turn provide a layer of data hiding for objects wishing to read or modify header fields of packets. The static class variable `offset_` is used to find the byte offset at which the mix header is located in an arbitrary packet. Two methods are provided to utilize this variable to access this header in any packet: `offset()` and `access()`. The latter is what we choose to access this particular header in a packet; the former is used by the packet header management class and is not used.
To access the mix packet header in a packet pointed by `p`, we simply say

`hdr_mix::access(p)`. The actual binding of `offset_` to the position of this header in a packet is done by routines inside `ns-packet.tcl` and `packet.cc`.
The static object `MIXHeaderClass` is used to provide linkage to OTcl when the mix header is enabled at configuration time. When the simulator executes, this static object object calls the `PacketHeaderClass` constructor with arguments `PacketHeader/MIX` and `sizeof(hdr_mix)`. This causes the size of the mix header to be stored and made available to the packet header manager at configuration time. The `bind_offset()` must be called in the constructor of this class, so that the packet header manager knows where to store the offset for this particular packet header.
By default ns includes all packet headers of all protocols in ns in every packet in the simulation. This is a lot of overhead. If the simulation is very packet-intensive, this could be a huge overhead. For instance the size of all protocols in ns is about 1.9KB; however, if we turn on only the needed headers we can reduce the size to less than 100 bytes. This reduction of unused packet headers can lead to major memory saving in large.scale traffic simulations.
To include only the packet headers that are used in the simulation, we type the following:

```
remove-all-packet-headers
add-packet-header RTP IP SR LL ARP Mac MIX
```

8.3 Route cache

Each node implementing *MixRoute* must maintain a route cache, containing routing information needed by the node. A node adds information to its route cache as its learns of new links between mix nodes in the ad hoc network; for example, a node may learn of new links when it receives a packet carrying a Mix Route Request (RREQ) or Mix Route Reply (MREP). Likewise, a node removes information from its route cache as it learns that existing links in the ad hoc network have broken; for example a node may learn of a broken link when it receives a packet carrying a Route Error or through the link-layer retransmission mechanism reporting a failure in forwarding a packet to its next-hop destination. Anytime a node adds new information to its route cache, the node checks each packet in its own send buffer to determine whether a route to that packet's destination now exists in the node's route cache. if so, the packet then is sent using that route and removed from the send buffer.
The route cache support storing more than one route to each destination. In searching the route cache for a route to some destination node, the route cache is indexed by destination node address.
Each implementation of *MixRoute* at any node searches its route cache and selects the best route to the destination from among those found. For example,

a node chooses the shortest route to the destination (the shortest sequence of hops).
The implementation of the route cache provide a fixed capacity for the cache. The following property describes the management of available space within a node's route cache:
In the implementation of *MixRoute* a appropriate policy for managing the entries in its route cache is used, such as when limited cache capacity requires a choice of which entries to retain in the cache. For example, the node uses the "least recently used" (LRU) cache replacement policy, in which the entry last used longest ago is discarded from the cache if a decision need to be made to allow space in the cache for some new entry being added.
Each entry in the route cache has a timeout associated with it, to allow that entry is deleted if not used within some time.
The implementation of the route cache consists of the class `MixRouteCache` which represents the route cache functionality.

Chapter 9

Simulation Model

We use the ns-2 simulation package to simulate a wireless ad hoc network, and implement MixRoute. At the physical layer, we simulate Lucent's WaveLAN card with a nominal bit rate of 2 Mbits/sec and a nominal transmission range of 250 meters. At the MAC layer, we use the distributed coordination function (DCF) of IEEE 802.11. At the network layer, the DSR routing protocol is used in routing of data packets. In our experiments, we simulate the stop-and-go Mix [17] where each Mix adds a random delay (uniformly distributed between 0 and 100 milliseconds) to each received packet before sending it out. At the beginning of each simulation run, a given number of randomly chosen nodes are designated as Mixes. The parameter values in our implementation of the Mix route algorithm are listed in Table 9.1. The network field is 1000m x 1000m with

ADVERTISEMENT_INTERVAL	3 secs
DREG_INTERVAL	3 secs
RUPD_INTERVAL	10 secs

Table 9.1: Parameter values in simulations

50 nodes initially uniformly distributed. *Random Way-point* mobility model [16] is used to generate node movement scenario. According to this model, a node travels to a random chosen location in a certain speed and stays for a while

before going to another random location. In our simulation, the maximum node speed varies from 0 to 20 m/sec, and the pause time is fixed to 30 seconds. Constant Bit Rate (CBR) sessions are used to generate data traffic. For each session, data packets of 512 bytes are generated in a rate of 4 packets per second. The source-destination pairs are chosen randomly from all the nodes (including Mixes). During 300 minutes simulation time, totally 25 sessions are scheduled with start times uniformly distributed between 0 and 180 seconds, and each session lasts for approximately 75 seconds.

Chapter 10

Evaluation

We want to evaluate the performance of the *MixRoute* algorithm in three aspects. First, we want to investigate the network performance of mix-net in a wireless ad hoc network. Two metrics are used:(i) *Packet delivery ratio* - the ratio between the number of data packets received and those originated by the sources. (ii) *Average end-to-end data packet latency* - the time from when the source generates the data packet to when the destination receives it. This includes: latency for determining mix route, network routing latency, cryptographic processing delays, queueing delay at the interface queue, retransmission delay at the MAC, propagation and transfer times. Second, we measure the average length of the mix route. When each mix has the same independent probability of being compromised, the probability that all mixes in a mix route be compromised decreases exponentially with the number of mixes. Third we evaluate the control overhead of the proposed algorithm. The metric we use is *normalized control packet load*, which is defined as the number of control packets transmitted per data packet delivered.

10.1 Running the simulation script

The test script is placed in the folder `ns-2.29/tcl/anonymity-mix`
The following files are part of the test script with `mixroute.tcl` as the main

file:

- `mixroute.tcl`
- `mix.tcl`
- `cbr-n50-mc10-r4`
- `scen-1000x1000-mix5`
- `scen-1000x1000-n50`

To run the tcl script the following command must be typed:
`ns mixroute.tcl`
The result of the typed command gives the following output to the terminal:

```
num_nodes is set to 55
Creating mobile nodes...
INITIALIZE THE LIST xListHead
Creating mixes...
Loading connection pattern...
Loading node scenario file...
Loading mix scenario file...
Load complete...
Starting Simulation...
end simulation
```

After the simulation process has ended there are generated some new files which can be found in the folder. These are `out.nam` and `mixroute.tr`. The file `out.nam` is opened inside `nam`. These can be done from a terminal using the command:
`nam out.nam &`
When nam has loaded the file the layout for the test is shown. This can be seen in Figure 10.1.
Unfortunately there were problems in the implemented protocol. Analyzing the tracefile `mixroute.tr` showed that specific nodes tried to send packets at the application layer, but the packets was not send to their destination. Thus the nodes would not communicate with each other. Due to lack of time the implemented protocol was not functioning properly because of some bugs that could not be detected.

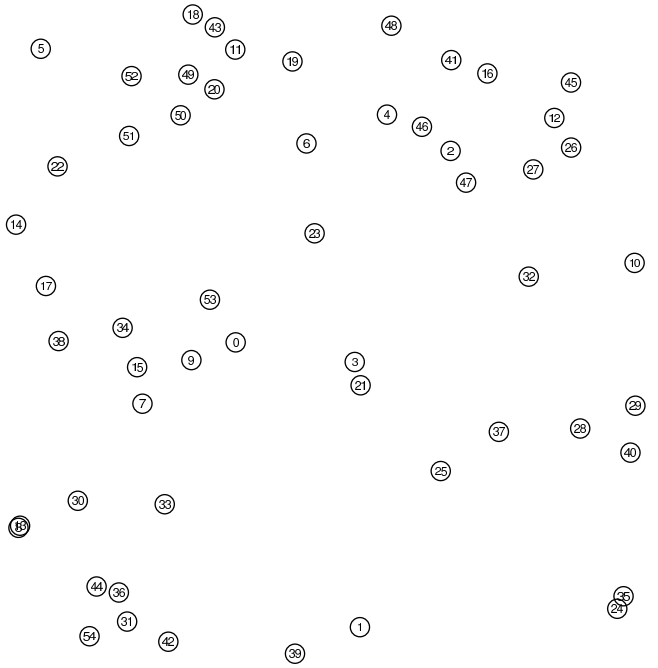

Figure 10.1: Screenshot from nam, showing the simulation with 55 nodes

10.2 Expected Simulation Results

The results should show that the proposed algorithm performs better than the static mix route algorithm by achieving higher packet delivery ratio and lower packet latency. These results should indicate that the proposed algorithm is adaptive to network topology change and ensure that data packets are routed along a short route in the physical network. In addition it should be shown that network performance suffers as node mobility increases, due to frequent change of mix route and large packet losses. We also wanted to show that both the static mix route algorithm and the proposed algorithm have lower packet latency in a high-mobility network than in the static network. The reason for this is that when network topology changes, the physical route for the same mix route is not constant and there is a good chance that some "distances" are short. We also wanted to show how the number of mixes in a network has effect on network performance. We would expect that more mixes in a network means shorter mix route and better network performance.
A plot of the average length of the proposed mix route as function of increasing

mobility and number of mixes, respectively should have shown that there is a linear correlation between the proposed mix route length and the number of mixes in the network. This is because mixes are randomly selected from the node set, and hence, uniformly distributed over the network area. We expect that high degree of anonymity can be achieved if there is a sufficient number of mixes in the network.
The overhead analysis of the proposed algorithm should be plotted with the Y-axis representing the ratio between the number of control packets transmitted and the number of data packets that are delivered, and the X-axis should represent the maximum node speed. The expected result of this plot is that the normalized control overhead is higher in a dynamic network than in a static network. When the traffic load is low the control overhead of the algorithm should be high. The reason is that, in our algorithm, mix advertisement packets are generated with no regard to data traffic load and set a lower bound for control traffic load. As the number of connections increases, the number of delivered packets should increase as well, which then drops the normalized control overhead.

Chapter 11

Conclusion

In this report, we have described new efforts in providing anonymous communication service in wireless ad hoc networks based on the mix-net scheme. We developed an efficient algorithm for determining mix route for an end-to-end connection. The design of the algorithm is based on two flooding processes: mix advertisement, and mix route discovery and update. Much efforts have been made to reduce transmission overhead in the two processes.
We have studied the problem of hiding source and destination information of packets in wireless ad hoc network. The background of this problem is the prevention of traffic analysis against the network. We argued that the existing security methods and schemas cannot be applied in wireless networks without degrading the network performance. To mitigate the performance degradation, we developed *MixRoute* with which the source node of a packet can dynamically determine the mix node to forward the packet. By executing a mix discovery protocol, the source node can find the mix node closest to it through which the route for the packet is the shortest one.
Unfortunately the implementation process of the protocol was very large and due to lack of time there were some bugs in the implementation that could not be found. The bugs were discovered during the evaluation of the implementation. The test script was implemented successfully, but the output of the tracefile showed that there were problems which was not solved. The nodes were not able to send the packets to their destination.
Overall the design of the protocol is an improvement to the existing mix-net

designs because of its dynamics, but it was not possible to show it through simulation results yet.
This project constitutes one of the first steps for providing anonymous communication in mobile ad hoc networks. In the future, the protocol could be debugged and made functional. Simulation results should then be used to compare the proposed algorithm to the static mix-net model to show the improvements.

Appendix A

Source Code

A.1 mixagent.h

```
1  #ifndef _MixAgent_h
2  #define _MixAgent_h
3  #include "mix_routecache.h"
4
5  #define MIX_BUFFER_CHECK 0.03 // seconds between buffer checks
6  #define MIX_SEND_TIMEOUT 30.0 // # seconds a packet can live in
       sendbuf
7  #define MIX_SEND_BUF_SIZE 64
8
9  #define ADVERTISE_INTERVAL  5.0
10 #define ROUTE_ADVERTISE_INTERVAL 15.0
11
12 struct MixSendBufEntry {
13   Time t;                          // insertion time
14   Packet* p;
15 };
16
17 class MixAgent;
18
19 class MixSendBufferTimer : public TimerHandler {
```

```
20 public:
21   MixSendBufferTimer(MixAgent *a) : TimerHandler() { a_ = a;}
22   void expire(Event *e);
23 protected:
24   MixAgent *a_;
25 };
26
27 class AdvertiseTimer : public TimerHandler {
28 public:
29   AdvertiseTimer(MixAgent *a) : TimerHandler() { a_ = a;}
30   void expire(Event *e);
31 protected:
32   MixAgent *a_;
33 };
34
35 class RouteAdvertiseTimer : public TimerHandler {
36 public:
37   RouteAdvertiseTimer(MixAgent *a) : TimerHandler() { a_ = a;}
38   void expire(Event *e);
39 protected:
40   MixAgent *a_;
41 };
42
43 struct MixEntry {
44         ns_addr_t addr_port_;
45         int seqno_;
46         int distance_;
47         int closest_mix_;
48         double expire_time_;
49         double changed_at; // when the best MADV last changed
50         double new_seqnum_at; // when we last heard a new seq number
51         double wst;     // running wst info
52         Event *reTx_event; // event used to schedule timeout action
53         Packet *pkt;     // MADV packet waiting to be sent
54         Event *exp_event;  // expire event
55 };
56
57 class MixAgentRetransmissionHandler : public Handler {
58   public:
59     MixAgentRetransmissionHandler(MixAgent *a_) { a = a_; }
60     virtual void handle(Event *e);
61   private:
62     MixAgent *a;
63 };
```

```
64
65  class MixExpireHandler : public Handler {
66    public:
67      MixExpireHandler(MixAgent *a_) { a = a_; }
68      virtual void handle(Event *e);
69    private:
70      MixAgent *a;
71  };
72
73  struct b {
74      Path routes[MAX_CACHE_SIZE];
75      int seqno;
76      bool mixroute_found;
77  };
78
79  class MixAgent: public Agent {
80  friend class MixSendBufferTimer;
81  friend class AdvertiseTimer;
82  friend class MixAgentRetransmissionHandler;
83  friend class RouteAdvertiseTimer;
84  friend class MixExpireHandler;
85  public:
86    MixAgent();
87    ~MixAgent();
88    int command(int argc, const char*const* argv);
89    void recv(Packet*, Handler*);
90    void handle(Event* e) { assert(target_); target_->recv((Packet*)e); }
91                            // this procedure handles delayed retransmission of
                                mix adv
92    void Terminate();
93  private:
94    int    mix_;
95    char   mix_alg_[32];
96    int    use_god_;
97    DSRAgent *dsr_agent_;          // used to install learned source route
98    MixEntry mix_list_[MAX_MIX_NUM];
99    int    closest_mix_count_;
100   MixExpireHandler *mix_expire_handler;
101   MixAgentRetransmissionHandler *reTx_handler;
102   RegEntry registration_table[MAX_NODE_NUM];
103   int    registration_count;
104   int    min_mixroute_len_;
105 //  struct b radv_cache[MAX_MIX_NUM];
106   int   dup_check[MAX_MIX_NUM][MAX_MIX_NUM];
```

```
107
108     /***** Mix List management *********/
109     MixEntry *findMix(Event *e);
110     void  updateClosestMix();
111     int   updateMixList(ns_addr_t a, int s, int d, double e);
112     ns_addr_t   selectMix();
113
114     /***** internal state of source *****/
115     MixSendBufEntry send_buf[MIX_SEND_BUF_SIZE];
116     RequestTable request_table;
117     MixSendBufferTimer send_buf_timer;
118     MixRouteCache route_cache[50];          // mix route cache
119
120     /****** internal state of mix ******/
121     int mix_advertise_num;
122     AdvertiseTimer advertise_timer;
123     int route_advertise_num;
124     RouteAdvertiseTimer route_advertise_timer;
125
126     /****** internal helper functions ******/
127     void stickPacketInSendBuffer(Packet* p);
128     void sendBufferCheck();
129     void sendOutPacketWithRoute(Packet *p, double delay);
130     int   sendOutPacketWithoutRoute(Packet *p, bool retry);
131     void getRouteForPacket(Packet* p, bool retry);
132     void sendRouteRequest(nsaddr_t target);
133     void sendRegistration();
134     void handleMixAdvertisement (Packet *p);
135     void handleRouteRequest(Packet *p);
136     void handleRegistration(Packet *p);
137     void handleRouteAdvertisement(Packet *p);
138     void acceptRoute(Path &reply_route, nsaddr_t dest, int NREG_seqno, int
            RADV_seqno, double latency);
139     void broadcastMixAdvertisement();
140     void broadcastRouteAdvertisement();
141     bool ignoreRouteAdvertisement(Path& route, int seq);
142
143     int route_request_num;
144     int  register_ ;
145     int registration_number;
146     double last_radv_broadcast;
147     int  last_registration ;
148 };
149
```

```
150 #endif
```

A.2 mixagent.cc

```
1  #include <object.h>
2  #include <float.h>
3  #include <tcp.h>
4  #include <ip.h>
5  #include <agent.h>
6  #include <packet.h>
7  #include <random.h>
8  #include <cmu-trace.h>
9  #include <dsr/path.h>
10 #include <dsr/constants.h>
11 #include <dsr/requesttable.h>
12 #include <dsr/dsragent.h>
13 #include <mix/hdr_mix.h>
14 #include <mix/mixagent.h>
15
16 #define MADV_STARTUP_JITTER 2.0 // secs to jitter start of periodic
       MADV from
17                                 // when start-mix msg sent to agent
18
19 #define alpha_ 0.875
20 #define wst0_ 1.0
21
22 static bool register_at_all_closest_mix = false;
23
24 /*=============================================================================
25   MixList management
26 -----------------------------------------------------------------------------
27 MixEntry *
28 MixAgent::findMix(Event *e)
29 {
30     int i;
31     //MAX_MIX_NUM in hdr_mix.h is set to 10
32     for(i = 0; i < MAX_MIX_NUM; i++) {
33         if( mix_list_[i].reTx_event == e || mix_list_[i].exp_event == e)
34             break;
35     }
36     return (&mix_list_[i]);
37 }
38
39 void
40 MixAgent::updateClosestMix()
```

```
41 {
42     double now = Scheduler::instance().clock();
43     int min_distance_ = 1000;
44     bool changed = false;
45
46     for(int i = 0; i < MAX_MIX_NUM; i++) {
47         if( mix_list_[i].expire_time_ > now &&
48                 mix_list_[i].distance_ < min_distance_)
49                     min_distance_ = mix_list_[i].distance_;
50     }
51
52     closest_mix_count_ = 0;
53     for(int i = 0; i < MAX_MIX_NUM; i++) {
54         if( mix_list_[i].expire_time_ > now &&
55                 mix_list_[i].distance_ == min_distance_) {
56
57             if( mix_list_[i].closest_mix_ == 0)
58                 changed = true;
59
60             mix_list_[i].closest_mix_ = 1;
61             closest_mix_count_++;
62         } else {
63             if( mix_list_[i].closest_mix_ == 1)
64                 changed = true;
65
66             mix_list_[i].closest_mix_ = 0;
67         }
68     }
69
70     if(changed && strcmp(mix_alg_, "MIX_Path") == 0) {
71         if( register_ == 1) {
72             int index = last_registration - MAX_NODE_NUM;
73             if( mix_list_[index].closest_mix_ == 0) {
74                 sendRegistration();
75             }
76         }
77     }
78 }
79
80 int
81 MixAgent::updateMixList(ns_addr_t a, int seq, int d, double e)
82 {
83     Scheduler & s = Scheduler::instance();
84     double now = s.clock();
```

```
85    int index = a.addr_ - MAX_NODE_NUM;
86
87    if( mix_list_ [index]. expire_time_ > now) {
88
89          if( mix_list_ [index]. seqno_ > seq) {
90             return -1;
91          }
92
93          if( mix_list_ [index]. seqno_ == seq) {
94            if( mix_list_ [index]. distance_ <= d) {
95               return -1;
96            }
97
98            // we've got a MADV without a new seq number
99            // this packet must have come along a different, but shorter, path
100           // than the previous one
101
102           mix_list_ [index]. distance_ = d;
103           mix_list_ [index]. expire_time_ = e;
104           mix_list_ [index]. changed_at = now;
105
106         } else {
107           // we've got a new seq number, end the measurement period
108           // for wst over the course of the old sequence number
109           // and update wst with the difference between the last
110           // time we changed the route (which would be when the
111           // best route metric arrives) and the first time we heard
112           // the sequence number that started the measurement period
113
114           mix_list_ [index]. wst = alpha_ * mix_list_ [index]. wst +
115                 (1.0 - alpha_) * ( mix_list_ [index]. changed_at -
                        mix_list_[index]. new_seqnum_at);
116
117           mix_list_ [index]. seqno_ = seq;
118           mix_list_ [index]. distance_ = d;
119           mix_list_ [index]. expire_time_ = e;
120           mix_list_ [index]. changed_at = now;
121           mix_list_ [index]. new_seqnum_at = now;
122         }
123   } else {
124         mix_list_ [index]. addr_port_ = a;
125         mix_list_ [index]. seqno_ = seq;
126         mix_list_ [index]. distance_ = d;
127         mix_list_ [index]. expire_time_ = e;
```

```
128             mix_list_[index].changed_at = now;
129             mix_list_[index].new_seqnum_at = now;
130             mix_list_[index].wst = wst0_;
131             mix_list_[index].pkt = 0;
132             mix_list_[index].reTx_event = 0;
133             mix_list_[index].exp_event = 0;
134         }
135
136         if(mix_list_[index].exp_event) {
137             s.cancel(mix_list_[index].exp_event);
138         } else {
139             mix_list_[index].exp_event = new Event;
140         }
141         s.schedule(mix_expire_handler, mix_list_[index].exp_event,
                mix_list_[index].expire_time_-now);
142
143         updateClosestMix();
144
145         if(mix_list_[index].closest_mix_ == 1)
146             return 0;
147         else
148             return -1;
149     }
150
151     void
152     MixExpireHandler::handle(Event *e)
153     {
154         Scheduler & s = Scheduler::instance();
155         MixEntry *m;
156
157         m = a->findMix(e);
158         assert(m);
159
160         m->seqno_ = -1;
161         m->distance_ = 0;
162         m->expire_time_ = 0;
163         m->changed_at = 0;
164         m->new_seqnum_at = 0;
165
166         if(m->reTx_event) {
167           s.cancel(m->reTx_event);
168           delete m->reTx_event;
169           m->reTx_event = 0;
170
```

```
171         Packet::free(m->pkt);
172         m->pkt = 0;
173       }
174
175       delete e;
176       m->exp_event = 0;
177
178       a->updateClosestMix();
179   };
180
181
182   ns_addr_t
183   MixAgent::selectMix()   // random
184   {
185     int index = Random::random() % closest_mix_count_;
186     int j = 0;
187
188     for(int i = 0; i < MAX_MIX_NUM; i++) {
189           if( mix_list_[i].expire_time_ <= Scheduler::instance().clock())
190               continue;
191
192           if( mix_list_[i].closest_mix_ == 1) {
193               if(j == index)
194                   return mix_list_[i].addr_port_;
195               else
196                   j++;
197           }
198     }
199     assert(0);
200   }
201
202   /*
203   char*
204   MixList::dump() const
205   {
206      static char buf[100];
207     MixEntry *p;
208     char *ptr = buf;
209
210     *ptr++ = '[';
211      p = head_;
212      while(p) {
213           ptr += sprintf(ptr, "%d(%d) ", p->addr_port_.addr_,
                    p->distance_);
```

```
214             p = p->next_;
215         }
216         *ptr++ = ']';
217         *ptr++ = '\0';
218         return buf;
219     }
220     */
221
222     /*=============================================================================
223         SendBuf management and helpers
224     -----------------------------------------------------------------------------*/
225     void
226     MixSendBufferTimer::expire(Event *)
227     {
228         a_->sendBufferCheck();
229         resched(MIX_BUFFER_CHECK + MIX_BUFFER_CHECK *
                Random::uniform(1.0));
230     };
231
232     void
233     MixAgent::stickPacketInSendBuffer(Packet* p)
234     {
235         Time min = DBL_MAX;
236         int min_index = 0;
237         int c;
238
239         for (c = 0 ; c < MIX_SEND_BUF_SIZE ; c ++)
240             if (send_buf[c].p == 0)
241             {
242                 send_buf[c].t = Scheduler::instance().clock();
243                 send_buf[c].p = p;
244                 return;
245             }
246             else if (send_buf[c].t < min)
247             {
248                 min = send_buf[c].t;
249                 min_index = c;
250             }
251
252         // kill somebody
253         drop(send_buf[min_index].p, DROP_PROXY_NOMIX);
254         send_buf[min_index].t = Scheduler::instance().clock();
255         send_buf[min_index].p = p;
256     }
```

```
257
258 void
259 MixAgent::sendBufferCheck()
260   // see if any packets in send buffer need route requests sent out
261   // for them, or need to be expired
262 { // this is called about once a second. run everybody through the
263   // get route for pkt routine to see if it's time to do another
264   // route request or what not
265   int c;
266
267   for (c = 0 ; c < MIX_SEND_BUF_SIZE ; c++) {
268           if (send_buf[c].p == 0) continue;
269
270           if (Scheduler::instance().clock() - send_buf[c].t >
                MIX_SEND_TIMEOUT) {
271                   drop(send_buf[c].p, DROP_PROXY_NOMIX);
272                   send_buf[c].p = 0;
273                   continue;
274           }
275
276           if(sendOutPacketWithoutRoute(send_buf[c].p, true) == 0)
277                 send_buf[c].p = 0;
278   }
279 }
280
281 /*============================================================
282   Route Request backoff
283 ------------------------------------------------------------*/
284 static bool
285 BackOffTest(Entry *e, Time time)
286 // look at the entry and decide if we can send another route
287 // request or not. update entry as well
288 {
289   Time next = ((Time) (0x1 << (e->rt_reqs_outstanding * 2))) *
          rt_rq_period;
290
291   if (next > rt_rq_max_period)
292           next = rt_rq_max_period;
293
294   if (next + e->last_rt_req > time)
295           return false;
296
297   // don't let rt_reqs_outstanding overflow next on the LogicalShiftsLeft 's
298   if (e->rt_reqs_outstanding < 15)
```

```
299             e->rt_reqs_outstanding++;
300
301   e->last_rt_req = time;
302
303   return true;
304 }
305
306 /*==========================================================================
307   Timer management and helpers
308 ----------------------------------------------------------------------------
309 void
310 AdvertiseTimer::expire(Event *)
311 {
312   a_->broadcastMixAdvertisement();
313   resched(ADVERTISE_INTERVAL * Random::uniform(0.75, 1.25));
314 }
315
316 void MixAgent::broadcastMixAdvertisement()
317 {
318     Packet *p = allocpkt();
319     hdr_ip *iph = hdr_ip::access(p);
320     hdr_mix *mixh = hdr_mix::access(p);
321     hdr_cmn *ch = hdr_cmn::access(p);
322
323     mixh->init();
324     mixh->type_ = MADV;
325     mixh->seqno_ = mix_advertise_num++;
326     mixh->lifetime_ = 3 * ADVERTISE_INTERVAL;
327     mixh->hop_count_ = 1;
328
329     iph->daddr() = IP_BROADCAST;
330     iph->dport() = here_.port_;
331
332     ch->size() = mixh->size();
333     ch->size() += IP_HDR_LEN; // dsragent will pass a broadcast packet
          directly
334                                 // to the link layer without changing its
                                       size
335     ch->next_hop_ = IP_BROADCAST;
336     ch->addr_type_ = NS_AF_INET;
337     ch->direction() = hdr_cmn::DOWN;
338     Scheduler::instance().schedule(this, p, Random::uniform(0.01));
339     //God::instance()->LogControl_mix(MADV, ch->size());
340 }
```

```
341
342 void
343 MixAgentRetransmissionHandler::handle(Event *e)
344 {
345     MixEntry *m;
346
347     m = a->findMix(e);
348     assert(m);
349
350     hdr_cmn *ch = hdr_cmn::access(m->pkt);
351     hdr_mix *mixh = hdr_mix::access(m->pkt);
352     mixh->hop_count_++;
353     ch->direction() = hdr_cmn::DOWN;
354     ch->next_hop() = IP_BROADCAST;
355     ch->addr_type() = NS_AF_INET;
356     Scheduler::instance().schedule(a, m->pkt, Random::uniform(0.01));
357     //God::instance()->LogControl_mix(MADV, ch->size());
358
359     // free this event
360     m->reTx_event = 0;
361     m->pkt = 0;
362     delete e;
363 };
364
365 void
366 RouteAdvertiseTimer::expire(Event *)
367 {
368     if(a_->registration_count > 0) {
369         a_->broadcastRouteAdvertisement();
370         resched(ROUTE_ADVERTISE_INTERVAL * Random::uniform(0.75,
                1.25));
371     }
372 }
373
374 static double total_interval = 0;
375 static int times = 0;
376
377 void
378 MixAgent::broadcastRouteAdvertisement()
379 {
380     Packet *p = allocpkt();
381     hdr_ip *iph = hdr_ip::access(p);
382     hdr_mix *mixh = hdr_mix::access(p);
383     hdr_cmn *ch = hdr_cmn::access(p);
```

```
384
385     mixh->init();
386     mixh->type_ = RADV;
387     mixh->seqno_ = route_advertise_num++;
388     mixh->appendToMixRoute(here_.addr_);
389
390     bcopy(registration_table, mixh->registration_table, sizeof(RegEntry) *
            MAX_NODE_NUM);
391     mixh->registration_count = registration_count;
392
393     iph->daddr() = IP_BROADCAST;
394     iph->dport() = here_.port_;
395
396     ch->size() = mixh->size();
397     ch->size() += IP_HDR_LEN; // dsragent will pass a broadcast packet
            directly
398                                     // to the link layer
399     ch->next_hop_ = IP_BROADCAST;
400     ch->addr_type_ = NS_AF_INET;
401     ch->direction() = hdr_cmn::DOWN;
402     ch->timestamp() = Scheduler::instance().clock();
403
404 //    Path route;
405 //    mixh->copyOutMixRoute(route);
406 //    ignoreRouteAdvertisement(route, mixh->seqno_);
407     int index = here_.addr_ - MAX_NODE_NUM;
408     dup_check[index][1] = mixh->seqno_;
409     Scheduler::instance().schedule(this, p, Random::uniform(0.01));
410     //God::instance()->LogControl_mix(RADV, ch->size());
411
412     if(last_radv_broadcast > 0) {
413         double interval = Scheduler::instance().clock() - last_radv_broadcast;
414          total_interval += interval;
415         times++;
416     }
417     last_radv_broadcast = Scheduler::instance().clock();
418 }
419
420 /*==========================================================================
421   MixAgent
422 ----------------------------------------------------------------------------
423 static class MixAgentClass : public TclClass {
424 public:
425   MixAgentClass() : TclClass("Agent/MixAgent") {}
```

```
426   TclObject* create(int, const char*const*) {
427     return (new MixAgent);
428   }
429 } class_MixAgent;
430
431 MixAgent::MixAgent(): Agent(PT_MIX), mix_(0), request_table(128),
        send_buf_timer(this),
432 advertise_timer(this), route_advertise_timer(this), min_mixroute_len_(1)
433 {
434   route_request_num = 1;
435   mix_advertise_num = 1;
436   registration_number = 1;
437   route_advertise_num = 1;
438   last_radv_broadcast = 0;
439    last_registration  = MAX_NODE_NUM;
440
441   reTx_handler = new MixAgentRetransmissionHandler(this);
442   mix_expire_handler = new MixExpireHandler(this);
443   bzero( mix_list_, sizeof(MixEntry) * MAX_MIX_NUM);
444   bzero( registration_table , sizeof(RegEntry) * MAX_NODE_NUM);
445   registration_count = 0;
446   register_ = 0;
447   bzero(dup_check, sizeof(int) * MAX_MIX_NUM * MAX_MIX_NUM);
448 /*
449   for(int i = 0; i < MAX_MIX_NUM; i++) {
450     radv_cache[i].seqno = 0;
451     radv_cache[i].mixroute_found = false;
452     for(int n = 0; n < MAX_CACHE_SIZE; n++) {
453       radv_cache[i].routes[n].reset();
454     }
455   }
456 */
457   Tcl& tcl = Tcl::instance();
458   tcl.eval("Simulator_set_mix_alg_");
459   strcpy(mix_alg_, tcl.result());
460   tcl.eval("Simulator_set_min_mixroute_len_");
461   min_mixroute_len_ = atoi(tcl.result());
462   assert(min_mixroute_len_ >= 1 && min_mixroute_len_ <=
        MAX_MIX_NUM);
463
464   use_god_ = 0;
465   tcl.eval("Simulator_set_use-god");
466   if(strcmp(tcl.result(), "ON") == 0)  use_god_ = 1;
467
```

```
468   for (int c = 0 ; c < MIX_SEND_BUF_SIZE ; c++)
469         send_buf[c].p = 0;
470 }
471
472 MixAgent::~MixAgent()
473 {
474 }
475
476 void MixAgent::Terminate()
477 {
478     int c;
479     for (c  = 0 ; c < MIX_SEND_BUF_SIZE ; c++) {
480         if (send_buf[c].p) {
481                 drop(send_buf[c].p, DROP_END_OF_SIMULATION);
482                 send_buf[c].p = 0;
483         }
484     }
485
486 //     if(here_.addr_ == 0)
487 //       printf("radv interval %f\n", total_interval/times);
488 }
489
490 int MixAgent::command(int argc, const char*const* argv)
491 {
492   TclObject *obj;
493
494   if (argc == 2)
495     {
496       if (strcasecmp(argv[1], "reset") == 0)
497         {
498           Terminate();
499           return Agent::command(argc, argv);
500         }
501       if (strcasecmp(argv[1], "set-as-mix") == 0)
502         {
503           mix_ = 1;
504           return TCL_OK;
505         }
506       if (strcasecmp(argv[1], "start") == 0)
507         {
508           if(mix_) {
509             // there is no traffic originated from mix, so the send_buf is
510             // not used in a mix
511
```

```
512                 if (!use_god_)
513                         advertise_timer.sched(Random::uniform(MADV_STARTUP_JITTER));
514             } else {
515                 // if closest_mix algorithm, it is not necessary to check
                        send_buf
516                 // because handleMixAdvertisement() will release the packets in
                        it
517
518                 if(strcmp(mix_alg_, "Closest_MIX") != 0) {
519                     send_buf_timer.sched(MIX_BUFFER_CHECK
520                                 + MIX_BUFFER_CHECK *
                                        Random::uniform(1.0));
521                 }
522             }
523             return TCL_OK;
524         }
525     }
526   else if (argc == 3)
527     {
528       if (strcasecmp(argv[1], "dsr-agent") == 0)
529         {
530           if( (obj = TclObject::lookup(argv[2])) == 0) {
531             fprintf(stderr, "MixAgent: %s lookup of %s failed\n", argv[1],
532                     argv[2]);
533             return TCL_ERROR;
534           }
535           dsr_agent_ = (DSRAgent*) obj;
536           return TCL_OK;
537         }
538     }
539   return Agent::command(argc, argv);
540 }
541
542 void MixAgent::recv(Packet* p, Handler*)
543 {
544   hdr_mix *mixh = hdr_mix::access(p);
545   hdr_ip *iph = hdr_ip::access(p);
546   hdr_cmn *ch = hdr_cmn::access(p);
547
548   if (mixh->valid_ != 1) {
549         // this must be a UDP packet
550         sendOutPacketWithoutRoute(p, false);
551   }
552   else if (mixh->valid_ == 1) {
```

```
553         if(mixh->type_ == MADV)
554                 handleMixAdvertisement(p);
555         else if(mixh->type_ == RREG)
556                 handleRegistration(p);
557         else if(mixh->type_ == RADV)
558                 handleRouteAdvertisement(p);
559         else {
560           if(mixh->dst_ == here_) {
561             // compare a variable ns_addr_t from struct mix_hdr, to the
                    variable from common/agent.cc here_ which is also type
                    ns_addr_t
562             // handle control packet receipt
563             assert(mixh->type_ == RREQ);
564             handleRouteRequest(p);
565           } else {
566             // handle forwarding
567             assert(mix_);
568             assert(mixh->mix_route_.len_ > 0);
569             iph->dst() = mixh->get_next_dst(here_.addr_, here_.port_);
570             iph->src() = here_;
571             ch->size() -= IP_HDR_LEN;
572             target_->recv(p, (Handler*) 0);
573           }
574         }
575   }
576 }
577
578 void
579 MixAgent::sendOutPacketWithRoute(Packet *p, double delay)
580   /* there must be a mix route in it */
581 {
582   hdr_mix *mixh = hdr_mix::access(p);
583   hdr_ip *iph = hdr_ip::access(p);
584   hdr_cmn *ch = hdr_cmn::access(p);
585
586   mixh->src_ = iph->src();
587   mixh->dst_ = iph->dst();
588   iph->dst() = mixh->get_next_dst(here_.addr_, here_.port_);
589   Scheduler::instance().schedule(this, p, delay);
590 }
591
592 int
593 MixAgent::sendOutPacketWithoutRoute(Packet *p, bool retry)
594   /* obtain a mix route to p's destination and send it off.
```

```
595         this should be a retry if the packet is already in the sendbuffer */
596 {
597   hdr_ip *iph = hdr_ip::access(p);
598   hdr_mix *mixh = hdr_mix::access(p);
599   hdr_cmn *ch = hdr_cmn::access(p);
600   ns_addr_t mix;
601   Path route;
602   God *god_;
603
604   if (iph->daddr() == here_.addr_) {
605         // it doesn't need a route, because it's for us
606         target_->recv(p, (Handler*)0);
607         return 0;
608   }
609
610   if(strcmp(mix_alg_, "Closest_MIX") == 0) {
611         if(closest_mix_count_ == 0) {
612           if(use_god_) {
613                 // use god info to build mix_list
614                 god_ = God::instance();
615                 for(int i = MAX_NODE_NUM; i < god_->nodes(); i++) {
616                       mix.addr_ = i;
617                       mix.port_ = 250;
618                       int d = god_->MinHops(here_.addr_, i);
619                       updateMixList(mix, 1, d,
                              Scheduler::instance().clock()+1000);
620                 }
621           } else {
622                 // I can only wait for MADV
623                 if (!retry) {
624                       stickPacketInSendBuffer(p);
625                 }
626                 return -1;
627           }
628         }
629
630         mix = selectMix();
631         mixh->init();
632         mixh->appendToMixRoute(mix.addr_);
633
634   } else {
635         int dest = iph->daddr();
636         if (!route_cache[dest].findRoute(route)) {
637           getRouteForPacket(p, retry);
```

```
638             return -1;
639         } else {
640             mixh->init();
641             mixh->installMixRoute(route);
642         }
643     }
644
645     assert(mixh->mix_route_.len_ > 0);
646     ch->size() += mixh->mix_route_.len_ * IP_HDR_LEN;
647     dsr_agent_->trace("%.9f %d -> %d MixRoute %s",
            Scheduler::instance().clock(),
648                               iph->saddr(), iph->daddr(), route.dump());
649     sendOutPacketWithRoute(p, 0.0);
650     return 0;
651 }
652
653 void
654 MixAgent::getRouteForPacket(Packet* p, bool retry)
655     /* try to obtain a route for packet
656        pkt is freed or handed off as needed, unless retry == true
657        in which case it is not touched */
658 {
659     hdr_ip *iph = hdr_ip::access(p);
660     Entry *e = request_table.getEntry(ID(iph->daddr(), ::IP));
661     Time time = Scheduler::instance().clock();
662
663     if (BackOffTest(e, time)) {
664         // it's time to start another route request cycle
665
666         if(closest_mix_count_ > 0) {
667             sendRouteRequest(iph->daddr());
668         }
669     }
670
671     if (!retry) {
672         stickPacketInSendBuffer(p);
673     }
674 }
675
676 void
677 MixAgent::sendRouteRequest(nsaddr_t target)
678     /* send a route request to "target" node */
679 {
680         if(closest_mix_count_ == 0)
```

```
681             return;
682
683
684             Packet *rrq = allocpkt();
685             hdr_ip *rriph = hdr_ip::access(rrq);
686             hdr_mix *rrmixh = hdr_mix::access(rrq);
687             hdr_cmn *rrcmh = hdr_cmn::access(rrq);
688
689             rrmixh->init();
690             rrmixh->type_ = RREQ;
691             rrmixh->seqno_ = route_request_num++;
692             ns_addr_t mix = selectMix();
693             rrmixh->appendToMixRoute(mix.addr_);
694             rriph->daddr() = target;
695             rriph->dport() = here_.port_;
696             rrcmh->size() = rrmixh->size();
697             rrcmh->timestamp() = Scheduler::instance().clock();
698             sendOutPacketWithRoute(rrq, 0.0);
699             //God::instance()->LogControl_mix(RREQ, rrcmh->size());
700 }
701
702 void
703 MixAgent::handleMixAdvertisement (Packet *p)
704 {
705         Scheduler & s = Scheduler::instance();
706         hdr_cmn *ch = hdr_cmn::access(p);
707         hdr_mix *mixh = hdr_mix::access(p);
708         hdr_ip *iph = hdr_ip::access(p);
709         double now = Scheduler::instance().clock();
710         MixEntry *m;
711
712         if (mix_) {
713             Packet::free(p);
714             return;
715         }
716
717         if(updateMixList(iph->src(), mixh->seqno_,
718                 mixh->hop_count_, mixh->lifetime_+now) < 0)
719         {
720             Packet::free(p);
721         } else {
722            // retransmit after certain delay
723            // first delete unsent old MADVs in queue
724             Packet* r;
```

```
725             while(r = dsr_agent_->ifq->prq_get_MADV(iph->src(),
                      mixh->seqno_)) {
726                   Packet::free(r);
727             }
728
729             int index = iph->src().addr_ - MAX_NODE_NUM;
730             if( mix_list_[index].reTx_event) {
731                   s.cancel( mix_list_[index].reTx_event);
732                   Packet::free( mix_list_[index].pkt);
733             } else {
734                   mix_list_[index].reTx_event = new Event;
735             }
736
737             mix_list_[index].pkt = p;
738             s.schedule(reTx_handler, mix_list_[index].reTx_event,
                      mix_list_[index].wst * 2);
739
740             if(strcmp(mix_alg_, "Closest_MIX") == 0) {
741                // see if the finding of the closest mix allows us to send out
742                // any of the packets we have waiting
743                for (int c = 0; c < MIX_SEND_BUF_SIZE; c++)
744                {
745                  if (send_buf[c].p == 0) continue;
746
747                  mixh = hdr_mix::access(send_buf[c].p);
748                  mixh->init();
749                  ns_addr_t mix = selectMix();
750                  mixh->appendToMixRoute(mix.addr_);
751                  ch = hdr_cmn::access(send_buf[c].p);
752                  ch->size() += IP_HDR_LEN;  // via one mix
753                  iph = hdr_ip::access(send_buf[c].p);
754                  dsr_agent_->trace("MixRoute_%d", mix.addr_);
755                  dsr_agent_->trace("%.9f_%d_->_%d_MixRoute_%d",
                           Scheduler::instance().clock(),
756                                        iph->saddr(), iph->daddr(),
                                             mix.addr_);
757                  sendOutPacketWithRoute(send_buf[c].p, 0.0);
758                  send_buf[c].p = 0;
759                }
760             }
761       }
762 }
763
764
```

```
765 void
766 MixAgent::handleRouteRequest(Packet *p)
767 {
768     assert (!mix_);
769
770     register_ = 1;
771
772     sendRegistration();
773
774     Packet::free(p);
775 }
776
777 void
778 MixAgent::sendRegistration()
779 {
780         Packet *rrp;
781         hdr_ip *rriph;
782         hdr_mix *rrmixh;
783         hdr_cmn *rrcmh;
784
785         if (closest_mix_count_ == 0)
786                 return;
787
788     if( register_at_all_closest_mix ) {
789       for(int i = 0; i < MAX_MIX_NUM; i++) {
790         if( mix_list_[i].closest_mix_ == 1) {
791             rrp = allocpkt();
792             rriph = hdr_ip::access(rrp);
793             rrmixh = hdr_mix::access(rrp);
794             rrcmh = hdr_cmn::access(rrp);
795
796             rrmixh->init();
797             rrmixh->type_ = RREG;
798             rrmixh->seqno_ = registration_number;
799             rriph->dst() = mix_list_[i].addr_port_;
800             rrcmh->size() = rrmixh->size();
801             rrcmh->timestamp() = Scheduler::instance().clock();
802             sendOutPacketWithRoute(rrp, 0.0);
803             //God::instance()->LogControl_mix(RREG, rrcmh->size());
804         }
805       }
806       registration_number++;
807     } else {
808         rrp = allocpkt();
```

```
809             rriph = hdr_ip::access(rrp);
810             rrmixh = hdr_mix::access(rrp);
811             rrcmh = hdr_cmn::access(rrp);
812
813             rrmixh->init();
814             rrmixh->type_ = RREG;
815             rrmixh->seqno_ = registration_number++;
816
817             int index = last_registration - MAX_NODE_NUM;
818             if( mix_list_ [index]. closest_mix_ == 1) {
819                 // use the old info
820                 rriph->daddr() = last_registration;
821                 rriph->dport() = 250;
822             } else {
823                 rriph->dst() = selectMix();
824                  last_registration = rriph->daddr();
825             }
826
827             rrcmh->size() = rrmixh->size();
828             rrcmh->timestamp() = Scheduler::instance().clock();
829             sendOutPacketWithRoute(rrp, 0.0);
830             //God::instance()->LogControl_mix(RREG, rrcmh->size());
831         }
832     }
833
834     void
835     MixAgent::handleRegistration(Packet *p)
836     {
837             hdr_cmn *ch = hdr_cmn::access(p);
838             hdr_mix *mixh = hdr_mix::access(p);
839             hdr_ip *iph = hdr_ip::access(p);
840             int n = iph->saddr();
841             bool changed = false;
842
843             assert(mix_);
844
845             if( registration_table [n]. seqno < mixh->seqno_) {
846                     registration_table [n]. seqno = mixh->seqno_;
847                     registration_count++;
848                     changed = true;
849             }
850
851             if(changed) {
852     //              if( route_advertise_timer . status () != TIMER_PENDING)
```

```
853                     route_advertise_timer.resched(0.0);
854             }
855
856             Packet::free(p);
857     }
858
859     bool
860     MixAgent::ignoreRouteAdvertisement(Path& route, int seq)
861     {
862     /*
863       int i;
864       int l = route.length();
865       int index = route[0].addr - MAX_NODE_NUM;
866
867       if(radv_cache[index].seqno > seq)
868             return true;
869
870       if(radv_cache[index].seqno < seq) {
871             //delete all
872             for(i = 0; i < MAX_CACHE_SIZE; i++) {
873                   if(radv_cache[index].routes[i].length() == 0)
874                     break;
875
876                   radv_cache[index].routes[i].reset();
877             }
878             radv_cache[index].seqno = seq;
879             radv_cache[index].mixroute_found = false;
880       }
881
882       if(l >= min_mixroute_len_) {
883             if(!radv_cache[index].mixroute_found) {
884                   radv_cache[index].mixroute_found = true;
885                   return false;
886             } else
887                   return true;
888       } else {
889           for(i = 0; i < MAX_CACHE_SIZE; i++) {
890             if(radv_cache[index].routes[i].length() == 0)
891                break;
892
893             if(radv_cache[index].routes[i] == route) {
894                return true;
895             }
896           }
```

```
897
898         assert(i < MAX_CACHE_SIZE);
899         radv_cache[index].routes[i] = route;
900         return false;
901     }
902 */
903 }
904
905 void
906 MixAgent::handleRouteAdvertisement(Packet *p)
907 {
908     hdr_cmn *ch = hdr_cmn::access(p);
909     hdr_mix *mixh = hdr_mix::access(p);
910     hdr_ip *iph = hdr_ip::access(p);
911     int index = iph->saddr() - MAX_NODE_NUM;
912     int l;
913     double latency = Scheduler::instance().clock() - ch->timestamp();
914     Path route;
915
916     // retrieve mix route
917     mixh->copyOutMixRoute(route);
918     route.reverseInPlace();
919
920     if (!mix_ && route.length() >= min_mixroute_len_) {
921
922         for(int i = 1; i < route.length(); i++) {
923             if(route[i] == route[i-1])
924                 printf("%s \n", route.dump());
925
926             assert (!(route[i] == route[i-1]));
927         }
928
929         for(int i = 0; i < MAX_NODE_NUM; i++) {
930             if(mixh->registration_table[i].seqno > 0) {
931                 acceptRoute(route, i, mixh->registration_table[i].seqno,
932 mixh->seqno_, latency);
933 /*
934 printf("%.9f %d -> %d add Route %s dseq %d rseq %d lt %f\n",
935 Scheduler::instance().clock(), here_.addr_, i, route.dump(),
936 mixh->registration_table[i].seqno, mixh->seqno_, latency);
937 */
938             }
939         }
940     }
```

```
941
942  /*
943    mixh->copyOutMixRoute(route);
944    if(ignoreRouteAdvertisement(route, mixh->seqno_)) {
945          Packet::free(p);
946          return;
947    }
948  */
949
950    l = mixh->mix_route_.len_;
951    if (l >= min_mixroute_len_)
952          l = min_mixroute_len_;
953
954    if(dup_check[index][l] < mixh->seqno_) {
955          for(int j = 1; j <= l; j++) {
956              if(dup_check[index][j] < mixh->seqno_)
957                    dup_check[index][j] = mixh->seqno_;
958          }
959    } else {
960          Packet::free(p);
961          return;
962    }
963
964    if(mix_) {
965          for(int i = 0; i < MAX_NODE_NUM; i++) {
966               if( registration_table[i].seqno > 0 &&
967                    registration_table[i].seqno <
                          mixh->registration_table[i].seqno) {
968                    registration_table[i].seqno = 0;
969                    registration_count--;
970               } else if(mixh->registration_table[i].seqno > 0 &&
971                    registration_table[i].seqno >
                          mixh->registration_table[i].seqno){
972                    // intermediate MIX may modify RADV packet
973                    mixh->registration_table[i].seqno = 0;
974               }
975          }
976
977          mixh->appendToMixRoute(here_.addr_);
978          if (++l >= min_mixroute_len_)
979                l = min_mixroute_len_;
980          dup_check[index][l] = mixh->seqno_;
981
982  //        mixh->copyOutMixRoute(route);
```

```
983 //       ignoreRouteAdvertisement(route, mixh->seqno_);
984   }
985
986   // retransmit
987    ch->direction() = hdr_cmn::DOWN;
988    ch->next_hop() = IP_BROADCAST;
989    ch->addr_type() = NS_AF_INET;
990    Scheduler::instance().schedule(this, p, Random::uniform(0.01));
991    //God::instance()->LogControl_mix(RADV, ch->size());
992 }
993
994 void
995 MixAgent::acceptRoute(Path &reply_route, nsaddr_t dest, int RREG_seqno,
        int RADV_seq,
996 double latency)
997   /* - enter the embedded mix route into our cache
998      - see if any packets are waiting to be sent out with this mix route
999      - free the pkt */
1000 {
1001   // add the new route into our cache
1002   assert(reply_route.length() > 0);
1003   int res = route_cache[dest].addRoute(reply_route, RREG_seqno,
          RADV_seq, latency);
1004   if(res == -1)
1005         return;
1006
1007   // back down the route request counters
1008   Entry *e = request_table.getEntry(ID(dest,::IP));
1009   e->rt_reqs_outstanding = 0;
1010   e->last_rt_req = 0.0;
1011
1012   // see if the addtion of this route allows us to send out
1013   // any of the packets we have waiting
1014   Time delay = 0.0;
1015   for (int c = 0; c < MIX_SEND_BUF_SIZE; c++)
1016     {
1017       if (send_buf[c].p == 0) continue;
1018
1019       hdr_ip *iph = hdr_ip::access(send_buf[c].p);
1020       hdr_mix *mixh = hdr_mix::access(send_buf[c].p);
1021       hdr_cmn *ch = hdr_cmn::access(send_buf[c].p);
1022       if (iph->daddr() == dest) {
1023           /* we need to spread out the rate at which we send packets
1024              in to the link layer to give ARP time to complete. */
```

```
1025
1026            mixh->init();
1027            mixh->installMixRoute(reply_route);
1028            dsr_agent_->trace("%.9f_%d_->_%d_MixRoute_%s",
                    Scheduler::instance().clock(),
1029                        iph->saddr(), iph->daddr(), reply_route.dump());
1030            ch->size() += mixh->mix_route_.len_ * IP_HDR_LEN;
1031            sendOutPacketWithRoute(send_buf[c].p, delay);
1032            delay += arp_timeout;
1033            send_buf[c].p = 0;
1034          }
1035      }
1036  }
```

A.3 hdr_mix.h

```
1  #ifndef _HdrMix_h
2  #define _HdrMix_h
3  #include <dsr/path.h>
4  #include <ip.h>
5
6  #define MIX_HDR_SZ          4       // the size of mix options header
7  #define MAX_NODE_NUM        50
8  #define MAX_MIX_NUM         10
9  #define DATA_PACKET_SIZE    512
10 #define MAX_MR_LEN          16
11
12 #define MADV                1       // mix advertisement
13 #define RREQ                2       // mix route request
14 #define RADV                3       // mix route advertisement
15 #define RREG                4       // node registration
16 #define MSOL                6       // mix solicitation
17 #define MREP                7       // mix reply
18
19 struct MixRoute {
20   nsaddr_t       addrs_[MAX_MR_LEN];
21   int            len_;
22   int            cur_index_;
23 };
24
25 struct RegEntry {
26   int    node;
27   int    seqno;
28 };
29
30 struct hdr_mix {
31         int  valid_;              /* is this header actually in the packet?
32                                      and initialized ? */
33
34         // the two fields used in data packet forwarding
35         ns_addr_t       src_;    // orginal source
36         ns_addr_t       dst_;    // saved packet destination
37         MixRoute        mix_route_;
38         int             type_;
39         int             seqno_;
40         double          rq_latency_;
41         int             hop_count_;      // MADV
```

```
42          Time            lifetime_ ;       // MADV
43          RegEntry        registration_table [MAX_NODE_NUM];
44          int             registration_count ;
45
46          static int  offset_ ;
47          inline static int& offset () { return offset_; }
48          inline static hdr_mix* access(const Packet* p) {
49                  return (hdr_mix*) p->access(offset_);
50          }
51
52          inline void copyOutMixRoute(Path& p) {
53                  p.reset ();
54                  p.setLength(mix_route_.len_);
55                  for (int i = 0 ; i < mix_route_.len_ ; i++)
56                      p[i] = ID(mix_route_.addrs_[i], ::IP);
57          }
58
59          inline void installMixRoute(const Path p) {
60                  mix_route_.len_ = p.length();
61                  mix_route_.cur_index_ = 0;
62                  for (int i = 0 ; i < p.length() ; i++)
63                      mix_route_.addrs_[i] = p[i].getNSAddr_t();
64          }
65
66          inline char* dump_path() {
67                  static char buf[100];
68                  char *ptr = buf;
69
70                  if ( mix_route_.len_ > 0 ) {
71                    *ptr++ = '[';
72                    for (int i = 0 ; i < mix_route_.len_ ; i++)
73                       ptr += sprintf(ptr, "%d ", mix_route_.addrs_[i]) ;
74                    *ptr++ = ']';
75                    *ptr++ = '\0';
76                    return buf;
77                  }
78          }
79
80          inline ns_addr_t get_next_dst(int my_addr, int mixagent_port) {
81                  if (mix_route_.addrs_[mix_route_.cur_index_] == my_addr)
82                          mix_route_.cur_index_++;
83
84                  if (mix_route_.cur_index_ == mix_route_.len_) {
85                          // this is the last mix
```

```
86                      return dst_;
87              } else {
88                      ns_addr_t   tmp;
89                      tmp.addr_ =
                            mix_route_.addrs_[mix_route_.cur_index_];
90                      tmp.port_ = mixagent_port;
91                      return tmp;
92              }
93      }
94
95      inline void appendToMixRoute(const nsaddr_t& a) {
96              assert(mix_route_.len_ < MAX_MR_LEN);
97              mix_route_.addrs_[mix_route_.len_++] = a;
98      }
99
100     inline void init() {
101             valid_ = 1;
102             type_ = 0x0000;
103
104             mix_route_.len_ = 0;
105             mix_route_.cur_index_ = 0;
106             bzero(mix_route_.addrs_, sizeof(nsaddr_t) * MAX_MR_LEN);
107             bzero( registration_table , sizeof(RegEntry) *
                        MAX_NODE_NUM);
108             registration_count = 0;
109     }
110
111     inline int size() {
112             int sz;
113
114             if (type_ == MADV || type_ == RREG || type_ == MSOL
                        || type_ == MREP) {
115                     sz = MIX_HDR_SZ + 4;
116             } if (type_ == RADV) { // variable size
117                     sz = MIX_HDR_SZ + 4 + registration_count * 6;
118             } else { // RREQ
119                     sz = DATA_PACKET_SIZE + mix_route_.len_ *
                                IP_HDR_LEN;
120             }
121
122             sz = ((sz+3)&(~3)); // align...
123             assert(sz >= 0);
124             return sz;
125     }
```

```
126 };
127
128 #endif
```

A.4 hdr_mix.cc

```
1  #include <stdio.h>
2  #include "hdr_mix.h"
3
4  int hdr_mix::offset_;
5  static class MIXHeaderClass : public PacketHeaderClass
6  {
7  public:
8    MIXHeaderClass() :
         PacketHeaderClass("PacketHeader/MIX",sizeof(hdr_mix))
9    {
10     bind_offset(&hdr_mix::offset_);
11   }
12 } class_mixhdr;
```

A.5 mix_routecache.h

```
1  #ifndef _Mix_RouteCache_h
2  #define _Mix_RouteCache_h
3  #include <dsr/path.h>
4  #include "hdr_mix.h"
5
6  #define MAX_CACHE_SIZE 50 // should be increased if you allow
       multi-route
7
8  struct a {
9    Path *routes_;
10   int route_seqno_;
11   double *latency_;              // route acquisition latency
12   double *weight_;
13 };
14
15 class MixRouteCache {
16 public:
17   MixRouteCache();
18   ~MixRouteCache();
19
20   int addRoute(const Path& route, int dest_seqno, int route_seqno, double
         latency);
21   // add this route to the cache (presumably we did a route request
22   // to find this route and don't want to lose it)
23
24   bool findRoute(Path& route);
25   // if there is a cached path from us to dest returns true and fills in
26   // the route accordingly. returns false otherwise
27
28   void deleteAll();
29
30   int seqno_;
31   int route_count_;
32 private:
33   struct a cache_[MAX_MIX_NUM];
34   double sum_;                   // sum of 1/latency_[i]
35 };
36
37 #endif
```

A.6 mix_routecache.cc

```
1  #include "mix_routecache.h"
2  #include "random.h"
3
4  /*==============================================================
5   Constructors
6  --------------------------------------------------------------*/
7  MixRouteCache::MixRouteCache()
8  {
9    for(int i = 0; i < MAX_MIX_NUM; i++) {
10          cache_[i].routes_ = 0;
11          cache_[i].latency_ = 0;
12          cache_[i].weight_ = 0;
13          cache_[i].route_seqno_ = 0;
14   }
15
16   seqno_ = 0;
17   sum_ = 0.0;
18   route_count_ = 0;
19 }
20
21 MixRouteCache::~MixRouteCache()
22 {
23   for(int i = 0; i < MAX_MIX_NUM; i++) {
24       if(cache_[i].routes_) {
25          delete[] cache_[i].routes_;
26          delete[] cache_[i].latency_;
27          delete[] cache_[i].weight_;
28       }
29   }
30 }
31
32 int MixRouteCache::addRoute(const Path& route, int dest_seqno, int
33 route_seqno, double latency)
34 {
35   int i;
36   int l = route.length();
37   int index = route[l-1].addr - MAX_NODE_NUM;
38
39   if(cache_[index].routes_ == 0) {
40          cache_[index].routes_ = new Path[MAX_CACHE_SIZE];
41          cache_[index].latency_ = new double[MAX_CACHE_SIZE];
```

```
42          cache_[index].weight_ = new double[MAX_CACHE_SIZE];
43
44          for (int n = 0; n < MAX_CACHE_SIZE ; n++) {
45                  cache_[index].routes_[n].reset();
46                  cache_[index].latency_[n] = 0;
47                  cache_[index].weight_[n] = 0;
48          }
49  }
50
51  if(dest_seqno < seqno_)
52          return -1;
53
54  if(dest_seqno > seqno_) {
55          deleteAll();
56          seqno_ = dest_seqno;
57  }
58
59  if(cache_[index].route_seqno_ > 0 && cache_[index].route_seqno_ <
        route_seqno) {
60          for(i = 0; i < MAX_CACHE_SIZE; i++) {
61                  if(cache_[index].routes_[i].length() == 0)
62                          break;
63
64                  cache_[index].routes_[i].reset();
65                  sum_ -= 1/cache_[index].latency_[i];
66                  cache_[index].latency_[i] = 0;
67                  cache_[index].weight_[i] = 0;
68                  route_count_--;
69          }
70  }
71
72  cache_[index].route_seqno_ = route_seqno;
73
74  // insert
75  for(i = 0; i < MAX_CACHE_SIZE; i++) {
76          if(cache_[index].routes_[i] == route)
77                  return 0;
78          if(cache_[index].routes_[i].length() == 0) {
79                  CopyIntoPath(cache_[index].routes_[i], route, 0,
                            route.length() - 1);
80                  cache_[index].latency_[i] = latency;
81                  sum_ += 1/cache_[index].latency_[i];
82                  route_count_++;
83                  break;
```

```
84          }
85      }
86
87      assert( i < MAX_CACHE_SIZE);
88
89      // calculate weight
90      for(int s = 0; s < MAX_MIX_NUM; s++) {
91          if(cache_[s].route_seqno_ == 0)
92              continue;
93
94          for(i = 0; i < MAX_CACHE_SIZE; i++) {
95              if(cache_[s].latency_[i] == 0)
96                  break;
97
98              cache_[s].weight_[i] = (1/cache_[s].latency_[i])/sum_;
99          }
100     }
101     return 0;
102 }
103
104 bool MixRouteCache::findRoute(Path& route)
105 {
106     if(route_count_ == 0)    return false;
107
108     double dice = Random::uniform(1.0);
109     double l = 0.0;
110
111     for(int s = 0; s < MAX_MIX_NUM; s++) {
112         if(cache_[s].route_seqno_ == 0)
113             continue;
114
115         for (int n = 0 ; n < MAX_CACHE_SIZE; n++) {
116             if(cache_[s].latency_[n] == 0)
117                 break;
118
119             if(dice <= l + cache_[s].weight_[n]) {
120                 route = cache_[s].routes_[n];
121                 return true;
122             } else
123                 l += cache_[s].weight_[n];
124         }
125     }
126     return false;
127 }
```

```
128
129 void MixRouteCache::deleteAll()
130 {
131   for(int s = 0; s < MAX_MIX_NUM; s++) {
132     if(cache_[s].route_seqno_ == 0)
133       continue;
134
135     for (int n = 0 ; n < MAX_CACHE_SIZE; n++) {
136       if(cache_[s].latency_[n] == 0)
137         break;
138
139       cache_[s].routes_[n].reset();
140       cache_[s].latency_[n] = 0;
141       cache_[s].weight_[n] = 0;
142     }
143     cache_[s].route_seqno_ = 0;
144   }
145
146   sum_ = 0;
147   route_count_ = 0;
148 }
```

Appendix B

Files for TCL script

B.1 mixroute.tcl

```
1  #default options
2  set opt(chan)          Channel/WirelessChannel    ;#channel type
3  set opt(prop)          Propagation/TwoRayGround ;#radio-propagation
       model
4  set opt(netif)         Phy/WirelessPhy            ;#network interface
       type
5  set opt(mac)           Mac/802_11                 ;#MAC type
6  set opt(ifq)           CMUPriQueue                ;#interface queue type
7  set opt(ifqlen)        50                         ;#max packet in ifq
8  set opt(ll)            LL                         ;#link layer type
9  set opt(ant)           Antenna/OmniAntenna        ;#antenna model
10 set opt(energy)        EnergyModel                ;#energy model is a
       node attribute. It initiates the node with an energy level.
11
12 set opt(x)             1000                       ;#X dimension of
       topography
13 set opt(y)             1000                       ;#Y dimension of
       topography
14 set opt(sc)            "scen-1000x1000-n50"       ;#Node scenario file
15 set opt(ms)            "scen-1000x1000-mix5"      ;#Mix scenario file
```

```
16 set opt(cp)              "cbr-n50-mc10-r4"
17
18 set opt(nn)              50              ;# number of nodes
19 set opt(seed)            0.0             ;# generates a new random number
       each time.
20 set opt(stop)            300               ;#300
21 set opt(tr)              "mixroute.tr"
22
23 # may be changed by command
24 set opt(nm)              5               ;# number of mixes
25 set opt(speed)           0               ;# speed of movement
26 set opt(mc)              10              ;# Max number of connections
27 set opt(run)             0               ;# run
28
29 #
       ===================================================================
30 #definition of variables
31
32 Simulator set AgentTrace_       ON
33 set RouterTrace                 OFF
34 Simulator set MacTrace_         OFF
35 Simulator set mixagent_port_    250
36 Simulator set mix_alg_          "MIX_Path"
37 Simulator set min_mixroute_len_ 1
38 Simulator set use-god           OFF
39
40 #datarates
41 Mac/802_11 set basicRate_      2Mb
42 Mac/802_11 set dataRate_       2Mb
43
44 #
       ===================================================================
45 proc usage { argv0 } {
46         puts "Usage: $argv0 -nm mixes -speed speed -mc conns -run
               run#"
47 }
48
49 proc getopt {argc argv} {
50         global opt
51         lappend optlist nm speed mc run
52
53         for {set i 0} {$i < $argc} {incr i} {
54                 set arg [lindex $argv $i]
55                 if {[string range $arg 0 0] != "-"} continue
```

```
56
57                  set name [string range $arg 1 end]
58                  set opt($name) [lindex $argv [expr $i+1]]
59          }
60  }
61
62  #
        ============================================================================
63  # Main Program
64  #
        ============================================================================
65  getopt $argc $argv
66  if { $opt(x) == 0 || $opt(y) == 0 } {
67          usage $argv0
68          exit 1
69  }
70
71  if {$opt(seed) > 0} {
72          puts "Seeding Random number generator with $opt(seed)\n"
73          ns-random $opt(seed)
74  }
75
76  source mix.tcl
77
78  #Remove all packet headers and then add the necessary
79  remove-all-packet-headers
80  add-packet-header RTP IP SR LL ARP Mac MIX
81
82  #
83  # Initialize Global Variables
84  #
85  set ns_         [new Simulator]
86  set chan        [new $opt(chan)]
87  set prop        [new $opt(prop)]
88  set topo        [new Topography]
89  set tracefd     [open $opt(tr) w]
90  $ns_ trace-all $tracefd
91
92  #Open the NAM trace file
93  set namfile [open out.nam w]
94  $ns_ namtrace-all $namfile
95
96
97  $topo load_flatgrid $opt(x) $opt(y)
```

```
98
99  $prop topography $topo
100
101 #
102 # Create God
103 #
104
105 #nn+nm = 55
106 set god_ [create-god [expr $opt(nn)+$opt(nm)]]
107
108 #$god_ log-parameter $opt(run) $opt(mc) $opt(speed) $opt(nm) 4
109
110 # create mobile nodes
111 puts "Creating mobile nodes..."
112 for {set i 0} {$i < $opt(nn) } {incr i} {
113         dsr-create-mobile-node $i
114         $god_ new_node $node_($i)
115         $node_($i) attach-mix-agent
116         $ns_ at 0.0 "$node_($i) start-mix"
117 }
118 puts "Creating mixes..."
119 # create mixes
120 for {set i 0} { $i < $opt(nm) } {incr i} {
121         set m   [expr $opt(nn) + $i]
122         dsr-create-mobile-node $m
123         $god_ new_node $node_($m)
124         $node_($m) attach-mix-agent
125         $node_($m) set-as-mix
126         $ns_ at 0.0 "$node_($m) start-mix"
127         set mixnode_($i) $node_($m)
128 }
129
130 #
131 # Source the Connection and Movement scripts
132 #
133 if { $opt(cp) == "" } {
134         puts "*** NOTE: no connection pattern specified."
135         set opt(cp) "none"
136 } else {
137         puts "Loading connection pattern..."
138         source $opt(cp)
139 }
140
141 if { $opt(sc) == "" } {
```

```
142         puts "*** NOTE: no node scenario file specified."
143         set opt(sc) "none"
144 } else {
145         puts "Loading node scenario file..."
146         source $opt(sc)
147 }
148
149 if { $opt(nm) == 0 } {
150         puts "*** NOTE: no mix configured."
151         set opt(ms) "none"
152 } else {
153         puts "Loading mix scenario file..."
154         source $opt(ms)
155         puts "Load complete..."
156 }
157
158 #
159 # Tell all the nodes when the simulation ends
160 #
161 for {set i 0} {$i < [expr $opt(nn) + $opt(nm)] } {incr i} {
162     $ns_ at $opt(stop).000000001 "$node_($i) reset-mix";
163     $ns_ at $opt(stop).000000002 "$node_($i) reset";
164 }
165 #$ns_ at $opt(stop).00000001 "puts \"NS EXITING...\" ; finish"
166 $ns_ at $opt(stop).00000001 "finish"
167
168 proc finish {} {
169         puts " finito... "
170         global ns_ tracefd namfile opt god_
171
172         $ns_ flush-trace
173         close $tracefd
174         close $namfile
175         exec nam out.nam &
176         $god_ show-stats-mix
177         $ns_ halt
178 }
179 #puts $tracefd "M 0.0 nn $opt(nn) nm $opt(nm) x $opt(x) y $opt(y)"
180 #puts $tracefd "M 0.0 sc $opt(sc) ms $opt(ms)"
181 #puts $tracefd "M 0.0 cp $opt(cp) seed $opt(seed)"
182
183 if {[Simulator set use-god] == "ON"} {
184         $god_ compute_route
185 }
```

```
186
187 puts "Starting Simulation..."
188 $ns_ run
```

B.2 mix.tcl

```
1  SRNode instproc attach-null-mix-agent {} {
2          $self instvar entry_point_ mixagent_
3
4          set mixagent_ [new Agent/NullMixAgent]
5          if { [Simulator set AgentTrace_] == "ON" } {
6              # do not want send trace after mix agent
7              Simulator set AgentTrace_ OFF
8              $self attach $mixagent_ [Simulator set mixagent_port_]
9              Simulator set AgentTrace_ ON
10         }
11         set entry_point_ $mixagent_
12 }
13
14 SRNode instproc mix-agent {} {
15         $self instvar mixagent_
16         return $mixagent_
17 }
18
19 SRNode instproc start-mix {} {
20         $self instvar mixagent_
21         $mixagent_ start
22 }
23
24 SRNode instproc reset-mix {} {
25         $self instvar mixagent_
26         $mixagent_ reset
27 }
28
29 SRNode instproc attach-closest-mix-agent {} {
30         global RouterTrace
31         $self instvar mixagent_ entry_point_ dsr_agent_
32
33         set mixagent_ [new Agent/ClosestMixAgent]
34         if { [Simulator set AgentTrace_] == "ON" } {
35             Simulator set AgentTrace_ OFF
36             $self attach $mixagent_ [Simulator set mixagent_port_]
37             Simulator set AgentTrace_ ON
38         } else {
39             $self attach $mixagent_ [Simulator set mixagent_port_]
40         }
41
```

```
42 #       dsragent will pass received MSOL's directly to the ClosestMix agent
43         $dsr_agent_ port-dmux $mixagent_
44
45         $mixagent_ dsr-agent $dsr_agent_
46         set drpT [cmu-trace Drop "MIX" $self]
47         $mixagent_ drop-target $drpT
48
49         set entry_point_ $mixagent_
50 }
51
52 SRNode instproc attach-mix-agent {} {
53         global RouterTrace
54          $self instvar mixagent_ entry_point_ dsr_agent_
55
56         set mixagent_ [new Agent/MixAgent]
57         if { [Simulator set AgentTrace_] == "ON" } {
58            Simulator set AgentTrace_ OFF
59             $self attach $mixagent_ [Simulator set mixagent_port_]
60            Simulator set AgentTrace_ ON
61         } else {
62             $self attach $mixagent_ [Simulator set mixagent_port_]
63         }
64
65 #       dsragent will pass received MADV's, RADV's directly to the
      MixAgent
66         $dsr_agent_ port-dmux $mixagent_
67
68         $mixagent_ dsr-agent $dsr_agent_
69         set drpT [cmu-trace Drop "MIX" $self]
70         $mixagent_ drop-target $drpT
71
72         set entry_point_ $mixagent_
73 }
74
75 SRNode instproc set-as-mix {} {
76          $self instvar mixagent_ dsr_agent_
77         $mixagent_ set-as-mix
78 #       $dsr_agent_ no-rreq-forwarding
79 }
```

B.3 cbr-n50-mc10-r4

```
1  #
2  # nodes: 50, max conn: 10, send rate: 0.25, seed: 426386
3  #
4  #
5  # 0 connecting to 1 at time 136.13853299810484
6  #
7  set source_(0) [new Agent/UDP]
8  $ns_ attach-agent $node_(0) $source_(0)
9  set dest_(0) [new Agent/UDP]
10 $ns_ attach-agent $node_(1) $dest_(0)
11 set cbr_(0) [new Application/Traffic/CBR]
12 $cbr_(0) set packetSize_ 512
13 $cbr_(0) set interval_ 0.25
14 $cbr_(0) set random_ 1
15 $cbr_(0) set maxpkts_ 10000
16 $cbr_(0) attach-agent $source_(0)
17 $ns_ connect $source_(0) $dest_(0)
18 $ns_ at 136.13853299810484 "$cbr_(0) start"
19 #
20 # 2 connecting to 3 at time 171.17716538308989
21 #
22 set source_(1) [new Agent/UDP]
23 $ns_ attach-agent $node_(2) $source_(1)
24 set dest_(1) [new Agent/UDP]
25 $ns_ attach-agent $node_(3) $dest_(1)
26 set cbr_(1) [new Application/Traffic/CBR]
27 $cbr_(1) set packetSize_ 512
28 $cbr_(1) set interval_ 0.25
29 $cbr_(1) set random_ 1
30 $cbr_(1) set maxpkts_ 10000
31 $cbr_(1) attach-agent $source_(1)
32 $ns_ connect $source_(1) $dest_(1)
33 $ns_ at 171.17716538308989 "$cbr_(1) start"
34 #
35 # 5 connecting to 6 at time 21.698187487990683
36 #
37 set source_(2) [new Agent/UDP]
38 $ns_ attach-agent $node_(5) $source_(2)
39 set dest_(2) [new Agent/UDP]
40 $ns_ attach-agent $node_(6) $dest_(2)
41 set cbr_(2) [new Application/Traffic/CBR]
```

```
$cbr_(2) set packetSize_ 512
$cbr_(2) set interval_ 0.25
$cbr_(2) set random_ 1
$cbr_(2) set maxpkts_ 10000
$cbr_(2) attach-agent $source_(2)
$ns_ connect $source_(2) $dest_(2)
$ns_ at 21.698187487990683 "$cbr_(2) start"
#
# 6 connecting to 7 at time 143.28203655000871
#
set source_(3) [new Agent/UDP]
$ns_ attach-agent $node_(6) $source_(3)
set dest_(3) [new Agent/UDP]
$ns_ attach-agent $node_(7) $dest_(3)
set cbr_(3) [new Application/Traffic/CBR]
$cbr_(3) set packetSize_ 512
$cbr_(3) set interval_ 0.25
$cbr_(3) set random_ 1
$cbr_(3) set maxpkts_ 10000
$cbr_(3) attach-agent $source_(3)
$ns_ connect $source_(3) $dest_(3)
$ns_ at 143.28203655000871 "$cbr_(3) start"
#
# 6 connecting to 8 at time 91.706512613085351
#
set source_(4) [new Agent/UDP]
$ns_ attach-agent $node_(6) $source_(4)
set dest_(4) [new Agent/UDP]
$ns_ attach-agent $node_(8) $dest_(4)
set cbr_(4) [new Application/Traffic/CBR]
$cbr_(4) set packetSize_ 512
$cbr_(4) set interval_ 0.25
$cbr_(4) set random_ 1
$cbr_(4) set maxpkts_ 10000
$cbr_(4) attach-agent $source_(4)
$ns_ connect $source_(4) $dest_(4)
$ns_ at 91.706512613085351 "$cbr_(4) start"
#
# 7 connecting to 8 at time 176.71679477985799
#
set source_(5) [new Agent/UDP]
$ns_ attach-agent $node_(7) $source_(5)
set dest_(5) [new Agent/UDP]
$ns_ attach-agent $node_(8) $dest_(5)
```

```
86  set cbr_(5) [new Application/Traffic/CBR]
87  $cbr_(5) set packetSize_ 512
88  $cbr_(5) set interval_ 0.25
89  $cbr_(5) set random_ 1
90  $cbr_(5) set maxpkts_ 10000
91  $cbr_(5) attach-agent $source_(5)
92  $ns_ connect $source_(5) $dest_(5)
93  $ns_ at 176.71679477985799 "$cbr_(5) start"
94  #
95  # 16 connecting to 17 at time 109.90308117582606
96  #
97  set source_(6) [new Agent/UDP]
98  $ns_ attach-agent $node_(16) $source_(6)
99  set dest_(6) [new Agent/UDP]
100 $ns_ attach-agent $node_(17) $dest_(6)
101 set cbr_(6) [new Application/Traffic/CBR]
102 $cbr_(6) set packetSize_ 512
103 $cbr_(6) set interval_ 0.25
104 $cbr_(6) set random_ 1
105 $cbr_(6) set maxpkts_ 10000
106 $cbr_(6) attach-agent $source_(6)
107 $ns_ connect $source_(6) $dest_(6)
108 $ns_ at 109.90308117582606 "$cbr_(6) start"
109 #
110 # 19 connecting to 20 at time 165.99053170810944
111 #
112 set source_(7) [new Agent/UDP]
113 $ns_ attach-agent $node_(19) $source_(7)
114 set dest_(7) [new Agent/UDP]
115 $ns_ attach-agent $node_(20) $dest_(7)
116 set cbr_(7) [new Application/Traffic/CBR]
117 $cbr_(7) set packetSize_ 512
118 $cbr_(7) set interval_ 0.25
119 $cbr_(7) set random_ 1
120 $cbr_(7) set maxpkts_ 10000
121 $cbr_(7) attach-agent $source_(7)
122 $ns_ connect $source_(7) $dest_(7)
123 $ns_ at 165.99053170810944 "$cbr_(7) start"
124 #
125 # 19 connecting to 21 at time 176.33500026368301
126 #
127 set source_(8) [new Agent/UDP]
128 $ns_ attach-agent $node_(19) $source_(8)
129 set dest_(8) [new Agent/UDP]
```

```
130 $ns_ attach−agent $node_(21) $dest_(8)
131 set cbr_(8) [new Application/Traffic/CBR]
132 $cbr_(8) set packetSize_ 512
133 $cbr_(8) set interval_ 0.25
134 $cbr_(8) set random_ 1
135 $cbr_(8) set maxpkts_ 10000
136 $cbr_(8) attach−agent $source_(8)
137 $ns_ connect $source_(8) $dest_(8)
138 $ns_ at 176.33500026368301 "$cbr_(8) start"
139 #
140 # 20 connecting to 21 at time 141.96586895825615
141 #
142 set source_(9) [new Agent/UDP]
143 $ns_ attach−agent $node_(20) $source_(9)
144 set dest_(9) [new Agent/UDP]
145 $ns_ attach−agent $node_(21) $dest_(9)
146 set cbr_(9) [new Application/Traffic/CBR]
147 $cbr_(9) set packetSize_ 512
148 $cbr_(9) set interval_ 0.25
149 $cbr_(9) set random_ 1
150 $cbr_(9) set maxpkts_ 10000
151 $cbr_(9) attach−agent $source_(9)
152 $ns_ connect $source_(9) $dest_(9)
153 $ns_ at 141.96586895825615 "$cbr_(9) start"
154 #
155 #Total sources/connections: 8/10
156 #
```

B.4 scen-1000x1000-n50

```
#
# nodes: 50, max x: 1000.00, max y: 1000.00
#
$node_(0) set X_ 376.049994774721
$node_(0) set Y_ 484.276860190925
$node_(0) set Z_ 0.000000000000
$node_(1) set X_ 569.037448232944
$node_(1) set Y_ 42.088477582299
$node_(1) set Z_ 0.000000000000
$node_(2) set X_ 709.706070744168
$node_(2) set Y_ 781.497091881790
$node_(2) set Z_ 0.000000000000
$node_(3) set X_ 560.909762203049
$node_(3) set Y_ 453.739032702921
$node_(3) set Z_ 0.000000000000
$node_(4) set X_ 610.911764919210
$node_(4) set Y_ 837.799351257799
$node_(4) set Z_ 0.000000000000
$node_(5) set X_ 73.390033856297
$node_(5) set Y_ 939.924410661747
$node_(5) set Z_ 0.000000000000
$node_(6) set X_ 486.010539133612
$node_(6) set Y_ 792.918200587618
$node_(6) set Z_ 0.000000000000
$node_(7) set X_ 231.395276992167
$node_(7) set Y_ 388.912527797233
$node_(7) set Z_ 0.000000000000
$node_(8) set X_ 38.999865323299
$node_(8) set Y_ 196.331301200416
$node_(8) set Z_ 0.000000000000
$node_(9) set X_ 307.392090311636
$node_(9) set Y_ 456.774696709853
$node_(9) set Z_ 0.000000000000
$node_(10) set X_ 995.333030361047
$node_(10) set Y_ 607.569922547448
$node_(10) set Z_ 0.000000000000
$node_(11) set X_ 375.474147754401
$node_(11) set Y_ 938.651763889838
$node_(11) set Z_ 0.000000000000
$node_(12) set X_ 870.804638398663
$node_(12) set Y_ 832.455659553124
```

```
42 $node_(12) set Z_ 0.000000000000
43 $node_(13) set X_ 41.309215964823
44 $node_(13) set Y_ 199.532856816155
45 $node_(13) set Z_ 0.000000000000
46 $node_(14) set X_ 35.099353944106
47 $node_(14) set Y_ 666.949884436460
48 $node_(14) set Z_ 0.000000000000
49 $node_(15) set X_ 223.061395668604
50 $node_(15) set Y_ 445.685728151033
51 $node_(15) set Z_ 0.000000000000
52 $node_(16) set X_ 766.803851699271
53 $node_(16) set Y_ 901.717277140635
54 $node_(16) set Z_ 0.000000000000
55 $node_(17) set X_ 81.608006724730
56 $node_(17) set Y_ 571.720533053826
57 $node_(17) set Z_ 0.000000000000
58 $node_(18) set X_ 309.499085269824
59 $node_(18) set Y_ 993.136474297472
60 $node_(18) set Z_ 0.000000000000
61 $node_(19) set X_ 464.269705016189
62 $node_(19) set Y_ 920.340705763285
63 $node_(19) set Z_ 0.000000000000
64 $node_(20) set X_ 343.135345581023
65 $node_(20) set Y_ 877.095736431869
66 $node_(20) set Z_ 0.000000000000
67 $node_(21) set X_ 569.987307898086
68 $node_(21) set Y_ 417.623913117166
69 $node_(21) set Z_ 0.000000000000
70 $node_(22) set X_ 99.639702524305
71 $node_(22) set Y_ 757.396693001155
72 $node_(22) set Z_ 0.000000000000
73 $node_(23) set X_ 498.547421697961
74 $node_(23) set Y_ 653.600248030585
75 $node_(23) set Z_ 0.000000000000
76 $node_(24) set X_ 968.489181354118
77 $node_(24) set Y_ 70.585656604221
78 $node_(24) set Z_ 0.000000000000
79 $node_(25) set X_ 694.391625335780
80 $node_(25) set Y_ 284.396593727752
81 $node_(25) set Z_ 0.000000000000
82 $node_(26) set X_ 896.700198881710
83 $node_(26) set Y_ 786.646248917659
84 $node_(26) set Z_ 0.000000000000
85 $node_(27) set X_ 838.051255632812
```

```
$node_(27) set X_ 752.672498942325
$node_(27) set Y_ 0.000000000000
$node_(28) set X_ 910.885346230155
$node_(28) set Y_ 350.562431597380
$node_(28) set Z_ 0.000000000000
$node_(29) set X_ 996.523481159677
$node_(29) set Y_ 385.900160592802
$node_(29) set Z_ 0.000000000000
$node_(30) set X_ 130.978451865613
$node_(30) set Y_ 238.381671622253
$node_(30) set Z_ 0.000000000000
$node_(31) set X_ 207.528550216448
$node_(31) set Y_ 50.846042012790
$node_(31) set Z_ 0.000000000000
$node_(32) set X_ 831.063104993926
$node_(32) set Y_ 586.335105113150
$node_(32) set Z_ 0.000000000000
$node_(33) set X_ 265.888309177190
$node_(33) set Y_ 232.964041283475
$node_(33) set Z_ 0.000000000000
$node_(34) set X_ 200.543970734148
$node_(34) set Y_ 506.729631312136
$node_(34) set Z_ 0.000000000000
$node_(35) set X_ 978.120392525811
$node_(35) set Y_ 90.175106599094
$node_(35) set Z_ 0.000000000000
$node_(36) set X_ 194.675591656129
$node_(36) set Y_ 95.992971669542
$node_(36) set Z_ 0.000000000000
$node_(37) set X_ 784.599899825822
$node_(37) set Y_ 345.331234584771
$node_(37) set Z_ 0.000000000000
$node_(38) set X_ 101.133219673231
$node_(38) set Y_ 486.206004659377
$node_(38) set Z_ 0.000000000000
$node_(39) set X_ 468.247840505920
$node_(39) set Y_ 0.866639469811
$node_(39) set Z_ 0.000000000000
$node_(40) set X_ 988.846072852608
$node_(40) set Y_ 312.952664469870
$node_(40) set Z_ 0.000000000000
$node_(41) set X_ 710.782262460029
$node_(41) set Y_ 922.318473887221
$node_(41) set Z_ 0.000000000000
```

```
$node_(42) set X_ 271.527901868756
$node_(42) set Y_ 19.533130820582
$node_(42) set Z_ 0.000000000000
$node_(43) set X_ 343.772543013256
$node_(43) set Y_ 973.227533146582
$node_(43) set Z_ 0.000000000000
$node_(44) set X_ 160.165603112999
$node_(44) set Y_ 105.201272266420
$node_(44) set Z_ 0.000000000000
$node_(45) set X_ 896.594692834582
$node_(45) set Y_ 887.598395492059
$node_(45) set Z_ 0.000000000000
$node_(46) set X_ 665.224448630280
$node_(46) set Y_ 818.958213865596
$node_(46) set Z_ 0.000000000000
$node_(47) set X_ 733.636201498176
$node_(47) set Y_ 732.096602971687
$node_(47) set Z_ 0.000000000000
$node_(48) set X_ 617.707240507730
$node_(48) set Y_ 975.761592844131
$node_(48) set Z_ 0.000000000000
$node_(49) set X_ 302.494517974290
$node_(49) set Y_ 899.711589361543
$node_(49) set Z_ 0.000000000000
```

B.5 scen-1000x1000-mix5

```
#
# mix nodes: 5, max x: 1000.00, max y: 1000.00
#
$node_(50) set X_ 290.523726592989
$node_(50) set Y_ 836.744627459646
$node_(50) set Z_ 0.000000000000
$node_(51) set X_ 210.612722627690
$node_(51) set Y_ 804.680690256316
$node_(51) set Z_ 0.000000000000
$node_(52) set X_ 214.428505767400
$node_(52) set Y_ 897.431510888449
$node_(52) set Z_ 0.000000000000
$node_(53) set X_ 336.203164637615
$node_(53) set Y_ 550.521968982315
$node_(53) set Z_ 0.000000000000
```

```
16 $node_(54) set X_ 149.289294623810
17 $node_(54) set Y_ 28.121087665014
18 $node_(54) set Z_ 0.000000000000
```

Bibliography

[1] Zero Knowledge Systems.
`http://www.freedom.net/`.

[2] Nicklas Beijar. Zone routing protocol (zrp).

[3] Oliver Berthold, Hannes Federrath, and Stefan Köpsell. Web MIXes: A system for anonymous and unobservable Internet access. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: Workshop on Design Issue in Anonymity and Unobservability*, pages 115–129. Springer-Verlag, LNCS 2009, 2000.

[4] Oliver Berthold, Hannes Federrath, and Stefan Köpsell. Web MIXes: A system for anonymous and unobservable Internet access. *Lecture Notes in Computer Science*, 2009:115–129, 2001.

[5] Justin Boyan. The anonymizer: Protecting user privacy on the web. *Computer-Mediated Communication Magazine*, 4(9), September 1997.

[6] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 4(2), February 1981.

[7] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[8] Wei Dai. Pipenet 1.1. Post to Cypherpunks mailing list, November 1998.

[9] Shlomi Dolev and Rafail Ostrovsky. Xor-trees for efficient anonymous multicast and reception. Technical Report 98-54, 23, 1998.

[10] H. Federrath, A. Jerichow, D. Kesdogan, and A. Pfitzmann. Security in public mobile communication networks, 1995.

[11] Michael J. Freedman, Emil Sit, Josh Cates, and Robert Morris. Introducing tarzan, a peer-to-peer anonymizing network layer. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, Cambridge, MA, March 2002.

[12] Eran Gabber, Phillip B. Gibbons, David M. Kristol, Yossi Matias, and Alain Mayer. On secure and pseudonymous client-relationships with multiple servers. *ACM Transactions on Information and System Security*, 2(4):390–415, 1999.

[13] Ian Goldberg and David Wagner. TAZ servers and the rewebber network: Enabling anonymous publishing on the world wide web. *First Monday*, 3(4), August 1998.

[14] C. Gulcu and G. Tsudik. Mixing E-mail with Babel. In *Network and Distributed Security Symposium - NDSS '96*. IEEE, 1996. `http://citeseer.nj.nec.com/2254.html`.

[15] M. Y. Sanadidi Mario Gerla Jiejun Kong, Xiaoyan Hong. Mobility changes anonymity: Mobile ad hoc networks need efficient anonymous routing. 2005.

[16] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[17] Dogan Kesdogan, Jan Egner, and Roland Büschkes. Stop-and-go MIXes: Providing probabilistic anonymity in an open system. In *Proceedings of Information Hiding Workshop (IH 1998)*. Springer-Verlag, LNCS 1525, 1998.

[18] J. Kong and X. Hong. Anodr: Anonymous on demand routing with untraceable routes for mobile ad-hoc networks, 2003.

[19] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet Package*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[20] David Mazières and M. Frans Kaashoek. The Design, Implementation and Operation of an Email Pseudonym Server. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS 1998)*. ACM Press, November 1998.

[21] Ulf Möller and Lance Cottrell. Mixmaster Protocol — Version 2. Unfinished draft, January 2000.

[22] C. Perkins. Ad hoc on demand distance vector (aodv) routing, 1997.

[23] Charles Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[24] Charles E. Perkins. Ad hoc networking. Addison-Wesley Publishing Company, 2001.

[25] A. Pfitzmann, B. Pfitzmann, and M. Waidner. ISDN-Mixes : Untraceable communication with small bandwidth overhead. In *GI/ITG Conference: Communication in Distributed Systems*, pages 451–463. Springer-Verlag, 1991.

[26] Andreas Pfitzmann and Marit K. Anonymity, unobservability, and pseudonymity - a proposal for terminology. In *International workshop on Designing privacy enhancing technologies*, pages 1–9, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[27] The VINT Project. The ns Network Simulator. Web Page. Available at `http://www-mash.CS.Berkeley.EDU/ns`.

[28] The proxomitron. http://www.proxomitron.cjb.net/.

[29] J. F. Raymond. Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: Workshop on Design Issue in Anonymity and Unobservability*, pages 10–29. Springer-Verlag, LNCS 2009, July 2000.

[30] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4), 1998.

[31] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[32] Martha Steenstrup. Cluster-based networks. pages 75–138, 2001.

[33] Elizabeth M. Belding-Royer Swaminathan Sundaramurthy. The ad-mix protocol for encouraging participation in mobile ad hoc networks. In *ICNP*, pages 156–167. IEEE Computer Society, 2003.

[34] Paul Syverson, Gene Tsudik, Michael Reed, and Carl Landwehr. Towards an Analysis of Onion Routing Security. In H. Federrath, editor, *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, pages 96–114. Springer-Verlag, LNCS 2009, July 2000.

[35] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.