Preface

This M.Sc. Thesis presents the results of my final master project entitled *Secure Routing in Mobile Ad Hoc Networks*. The master project was carried out in the period 1st of July to 31st of December 2003 and corresponds to 30 ETCS points. It has been the final part of my studies for the Master of Science degree (In Danish Civilingeniør-uddannelsen) at the Department of Informatics and Mathematics Modelling, IMM, Technical University of Denmark, DTU.

Associate Professor Christian Damsgaard Jensen, IMM, was supervisor of my M.Sc. Thesis project.

I would like to take this opportunity to thank Christian Damsgaard Jensen for his counseling, support and not least his interest in my M.Sc. Thesis project.

Furthermore, I would like to thank the following persons: Casper Borly, Finn Conrad, Simon Haldrup and Tom Skovgaard for comments and discussions during my thesis work.

Last but not least, I would like to thank my girlfriend Lene Skovgaard for her patience and understanding.

Lyngby, DTU, 30th December 2003

Lennart Conrad

Abstract

This M.Sc. Thesis has focused on the design and implementation of trust based route selection in mobile wireless ad hoc networks. A system that stores and updates trust values for nodes encountered in mobile ad hoc networks is designed and implemented. The trust values are used to base routing decisions on. Since a route consists of many nodes that are grouped, different strategies for evaluation of routes based of the nodes trust values have been designed and implemented.

The implemented system has been integrated with an existing implementation of the Dynamic Source Routing protocol; A protocol, used for communication in mobile wireless ad hoc networks. To identify how and where to incorporate the trust in the DSR protocol an analysis of possible malicious attacks against the protocol have been conducted and presented.

To evaluate the performance impacts of applying trust based route selection to the DSR protocol several different simulations have been performed on the Ns-2 network simulator. The results from these simulations have been analyzed and presented.

Abstract in Danish

Denne M.Sc Thesis har fokuseret på design og implementering af tillids baserede rute valg i mobile trådløse ad hoc netværk. Et system som gemmer og opdaterer tillids værdier for kendte noder i ad hoc netværk er blevet designet og implementeret. Tillids værdierne benyttes til at basere rute valg på. Eftersom en rute består af mange noder som er grupperede, er forskellige strategier til at evaluere ruter, på baggrund af nodernes tillids værdier, blevet designet og implementeret.

Det implementerede system er blevet integreret med en eksisterende implementering af ”Dynamic Source Routing” protokollen, en protokol som benyttes til kommunikation i trådløse ad hoc netærk. For at indentificere hvor og hvordan tillid har kunnet indbygges i DSR protokollen, er der foretaget og præsenteret en analyse af mulige angreb på protokollen.

For at evaluere effekten af at benytte tillids baserede rute valg i DSR er der blevet foretaget flere forskellige simuleringer på Ns-2 netværks simulatoren. De opnåede resultater her af, er blevet analyserede og presenterede.

1 Introduction

The need of access to wireless communication is increasing rapidly as the desire of mobile connectivity with devices such as cell phones, PDA's and laptops to data networks is becoming more and more prevalent
The growing demand for mobile access is reflected by the current increase in the establishment of local wireless networks and the construction of new pylons for tele communication. However several factors such as, geographic conditions, economics, and spontaneous occurring needs can make the establishment or existence of access points impossible.

To manage situations where access points are out of transmission range, *ad hoc routing protocols* that can be used in mobile wireless networks have been developed.

An *ad hoc network* is a collection of wireless mobile nodes that dynamically functions as a network without the use of any existing infrastructure and centralized administration. Nodes move around which can cause links to be broken and established. Due to the relatively short transmission range of wireless devices, nodes in the network collaborate to route data to destinations that might be out of the senders transmission range. An ad hoc network is illustrated in Figure 1-1. As illustrated all nodes are not in direct connection with each other but can use other nodes as relays in order to transmit to a destination.
The figure also illustrates another important property of the shown ad hoc network, the inaccessibility to servers or centralized administration.

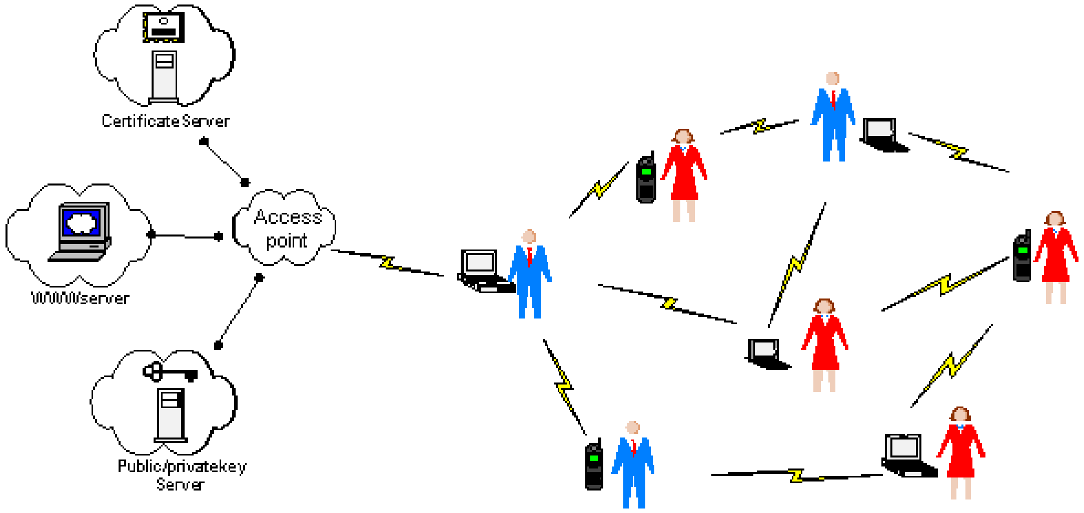


Figure 1-1: Mobile wireless ad hoc network

Ad hoc networks are often proposed for search and rescue mission and military operations where existing infrastructure have been damaged and is inoperative. In such situations the nodes are related by outside factors such as organizational hierarchies.

Another type of ad hoc networks, often referred to, as *collaborative networks* are networks where agents with no common relations join together to achieve their own personal goal of sending packets to a destination.

The structure of collaborative ad hoc networks is untraditional, since nobody can claim ownership and control of the network and thereby attend to administration of the network and require payment for its use.

The structure of the ad hoc network gives rise to security issues of different severity, since malicious nodes can seek to exploit the openness of the network.

This M.Sc thesis investigates several existing security solutions for ad hoc networks and proposes a trust based route selection solution. The solution addresses the problem that occurs when malicious *nodes* starts to drop packets they were supposed to forward. The design and implementation of the solution is presented and results from simulations with the implemented system is analyzed and discussed.

Due to the technical nature of this M.Sc. thesis, it is expected that readers of this have a general technical insight and knowledge of software development methods.

1.1 Motivation

Most common protocols for mobile wireless networks build on the assumptions that nodes in the network are willing to participate to the networks existence by forwarding packets for other nodes.

In general most mobile devices operate on battery power, which means that each transmission has a cost in terms of power consumption. This results in a conflict, since nodes have to perform the task of forwarding data, from which they achieve no benefits and as a result consume their own battery power. There is little reason to assume that some nodes will not try to achieve the benefits of participating in the network and avoid the disadvantages it involves. This could mean that some nodes refuse to forward packets as supposed and thereby decrease the efficiency of the network. Because of the nature of the ad hoc network it is difficult to identify nodes that express such malicious behavior, because the node originating the transmission might be out of range to detect the malicious act.

The open structure, lack of existing infrastructure and un-accessibility to trusted servers make traditional security methods and system insufficient for application in mobile wireless ad hoc networks. Achieving different levels of security therefore represents a major issue for the distribution and use of ad hoc networks.

By allowing an *unknown node* to forward data, nodes perform a trust-based decision. Trust is a well-known sociological concept that humans on a daily basis base decision on. By incorporating trust in ad hoc routing protocols and thereby mimicking human behavior, it is expected that the establishment and evolution of trust can be used to detect nodes that betrays the trust placed in them. The detection of untrustworthy nodes can be used to apply trust based route selection strategies to ad hoc routing protocols and thereby increase the effectiveness of the network.

1.2 Aim and Objectives

This section states the aim and objectives that have been stated for the Master project. Because of the exploratory nature of the project the list of objectives has evolved in order to reflect the increasing knowledge and insight in the areas of ad hoc routing and trust.

The aim of the assignment has been to design and implement a system that can be used for trust based routing. The system must make it possible for nodes to store and updates crisp values that represent their trust in other nodes. These values should be adjusted based on the experiences the nodes have. When a route is selected is must be selected by an evolution of the nodes on the routes values. It is the aim that such a system can be applied to the DSR protocol to achieve route selection strategies that can avoid nodes with low values.

The primary objective, which is stated in Figure 1-2 has been clear from the start.

> ***Primary objective****: To apply trust based route selection to the Dynamic Source Routing (DSR) protocol, in order to fortify the protocol and improve route selection, which can increase throughput in situations where malicious nodes are present in the network.*

Figure 1-2: Primary objective.

In order to achieve the primary objective several intermediate objectives are defined. These objectives are divided into three categories: preliminary, main and post objectives

1.2.1 Preliminary objectives

The preliminary objectives are objectives that need to be fulfilled in order to gain the sufficient knowledge to fulfill the main objectives.

PRE-O 1. **Examine existing ad hoc routing protocols**. To attain knowledge and insight in the area of ad hoc routing existing protocols must be examined.

PRE-O 2. **Investigate security solutions applied to ad hoc networks and ad hoc routing protocols.** Existing security solutions that have been applied to ad hoc networks and routing protocols must be investigated, to identify useful methods and approaches.

PRE-O 3. **Research the area of trust**. Research the area of trust management and trust in general to gain deeper knowledge of trust as a concept and to find formal methods for expressing trust.

1.2.2 Main objectives

The fulfillment of the main objectives leads to the fulfillment of the primary objective.

M-O 1. **Analyze the DSR protocol**. To detect weaknesses and possible pitfalls the DSR protocol [Johnson1] must be analyzed. Furthermore, situations where trust can be incorporated must be identified.

M-O 2. **Design and implement components to incorporate trust based route selection to the existing DSR protocol**. Based on the analysis modules that incorporate trust must be designed, implemented and integrated with the existing implementation of the DSR protocol. Particular emphasis must be put on the design of different routing strategies.

1.2.3 Post objective

This objective needs to be fulfilled to determine to what extend the primary objective has been fulfilled and to identify possible areas of improvement.

P-O 1. **Simulate behavior of the implemented trust based routing and analyze results**. Simulations with the implemented extension have to be carried out. The results must be analyzed to determine the impact of applying the trust based routing strategies and to detect possible areas of improvements.

1.3 Report Roadmap

This section gives a brief presentation of each chapter in the thesis.

Chapter 1 Introduction: In this chapter an introduction to the investigated area ad hoc networks is presented. Furthermore, the motivation and the objectives are presented.

Chapter 2 State of the art: This chapter concerns examined areas such as:

- Ad hoc routing protocols
- Trust management systems
- Security in Ad hoc networks
- Trust

Chapter 3 Analysis: In this chapter an analysis of the DSR protocol is presented. The analysis focuses on how to apply trust to the protocol and how malicious nodes can misuse the protocol.

Chapter 4 Design: The design of the component that is used to incorporate trust based route selection in DSR is described. Furthermore the integration with the existing DSR classes is covered.

Chapter 5 Implementation and tests: This chapter covers the implementation of the designed system and discusses the tests that have been performed. Furthermore, it gives a small introduction to the Ns-2 network simulator.

Chapter 6 Simulations and Results: In this chapter the performed simulations are described and the achieved results are discussed and analyzed.

Chapter 7 Future Work, Improvements: Some of the future area of work and possible areas of improvements are discussed in this chapter.

Chapter 8 Conclusion: The final chapter presents the conclusion and contribution of the project and summarizes the achieved results.

2 State of the art

To achieve the preliminary objectives, PRE-O 1 and PRE-O 3, several areas such as ad hoc networks, routing protocols, trust and security in ad hoc networks has been researched. This chapter investigates these areas.

2.1 Introduction to Ad Hoc Networks and Routing

An ad hoc network consists of mobile wireless nodes. Nodes participating in the network manage routing without the use of any existing infrastructure. Further more no centralized access point exists in the network. Mobile wireless nodes will typically have limited transmission range, which means that packets might have to be forwarded by several nodes in order to get from one node in the network to another. Figure 2-1 below illustrates how node A uses a route through node B to get data to node C, because C is out of A's transmission range.

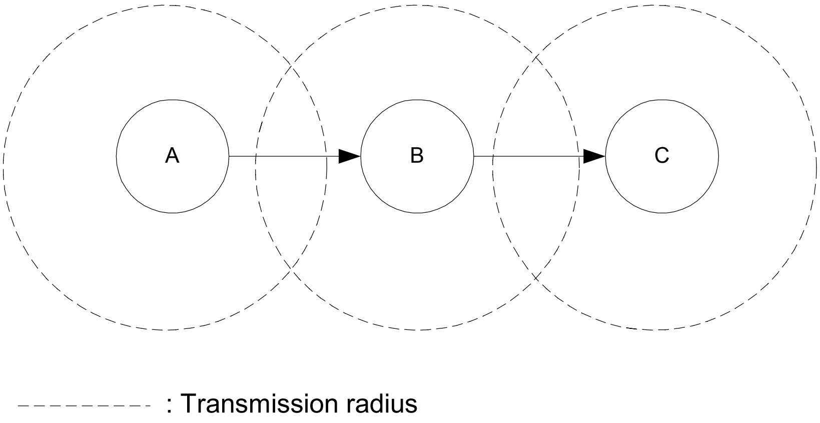


Figure 2-1: Node A transmits a package to node C by routing it through node B.

Routing protocols for ad hoc networks need to account for several aspects. Since nodes can move around, and enter and leave the network, the network topology can change rapidly. Due to the possible rapid changes in topology, it can require a lot of communication for a node to keep a static picture of the topology. Since the nodes are mobile they operate on battery power, which limits the amount of data they can transmit, before recharging is necessary. Furthermore, the possible bandwidth for mobile devices today is not as high as for stationary networks. Most mobile devices today have less processing power and memory than standard PC's.

Due to these circumstances ad hoc routing protocols must minimize the number of packets used for maintaining the routes and must be able to adapt to changes in the topology.

Another issue in ad hoc networks is that links between nodes are not always bidirectional but can be unidirectional. As illustrated in Figure 2-2 below, this means that even though node A can transmit to node C through node B, it is not sure that node C can use the same route back to A. Several issues such that low battery power or different hardware types can cause unidirectional links.

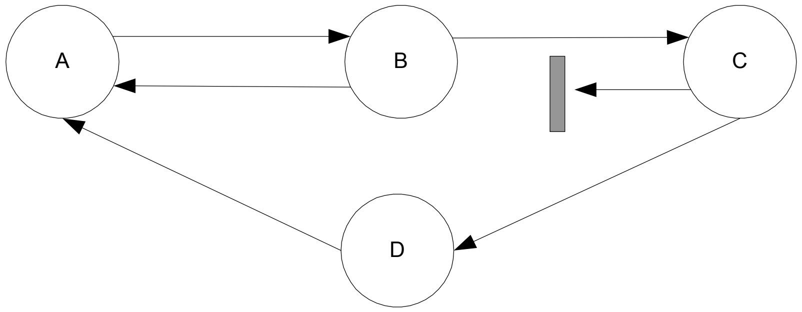


Figure 2-2: Unidirectional links, node A can transmit to node B and B to C, but node C cannot transmit to node B and must use a different route to A.

If node C should be able to transmit through the node B, the distance between the two nodes should be decreased. Bi-directional links can be established by lowering the distance between nodes.

The remainder of this chapter includes a general introduction to ad hoc routing protocols and an introduction to four well-known ad hoc routing protocols. The mode of operation for each protocol is explained. Some comparisons of the performance of these protocols have been conducted and described in the literature and some of the results from these comparisons are summarized.

2.1.1 Routing protocols

There are several different principles that can be applied when constructing routing protocols for ad hoc networks. This section introduces some of these principles.

Proactive vs. Reactive

Protocols can be *proactive* (also called *table driven*) which means that nodes periodically registers changes in the topology and updates routing information. The routes are stored and maintained in routing tables. Proactive protocols have the advantage that there is little latency since routes are already available [Zou], but the disadvantage that they require nodes to periodically update routing tables. In a highly dynamic network this increases routing related traffic. The opposite approach is the *reactive* (also called *on-demand*) protocols. Routes are first discovered on demand, when data needs to be transmitted to a node where no route has yet been discovered. The major advantage of on demand routing is that it saves bandwidth because it limits the routing overhead. The disadvantage is the latency at the beginning of transmission to nodes when no route, have yet been discovered [Zou].

Source routing vs. Hop-by-hop routing

Some routing protocols include the entire route in the packet header. This type of routing is referred to as source routing. It has the advantage that intermediate nodes are not required to maintain up-to-date routing information to forward the packet. The disadvantage of source routing is that the packet size can grow, especially in large networks.

Hop-by-hop routing protocols, such as vector protocols, only include information about the destination in the header and use local tables to determine the next hop on the route. This has the advantage that it limits the packet size, but has the disadvantage that it requires nodes to maintain and exchange routing information.

In the following sections some routing protocols that build on the described principles are investigated. Particular emphasis will be put on the description of the DSR protocol since this is the foundation for the subsequent design and implementation.

2.1.2 The Destination-Sequenced Distance Vector (DSDV) Protocol

The Destination-Sequenced Distance Vector protocol (DSDV) was introduced by Charlie E. Perkins and Elizabeth Royer in 1994 [Perkins1], [Guoyou1]. The protocol is a modification of the Bellman-Ford routing algorithm [Siek1]. This is a decentralized routing algorithm, which requires that each router inform its neighbours of its routing table. These modifications make the protocol more suitable for routing in ad hoc networks. The protocol is of the hop-by-hop type.

Assumptions

A1. DSDV assumes that all links in the network are bi-directional.

Mode of operation

DSDV operates by having each node maintain a table with information about distances and information about the next node on a route. The protocol can be explained by looking at a small topology, such as the one illustrated in Figure 2-3.

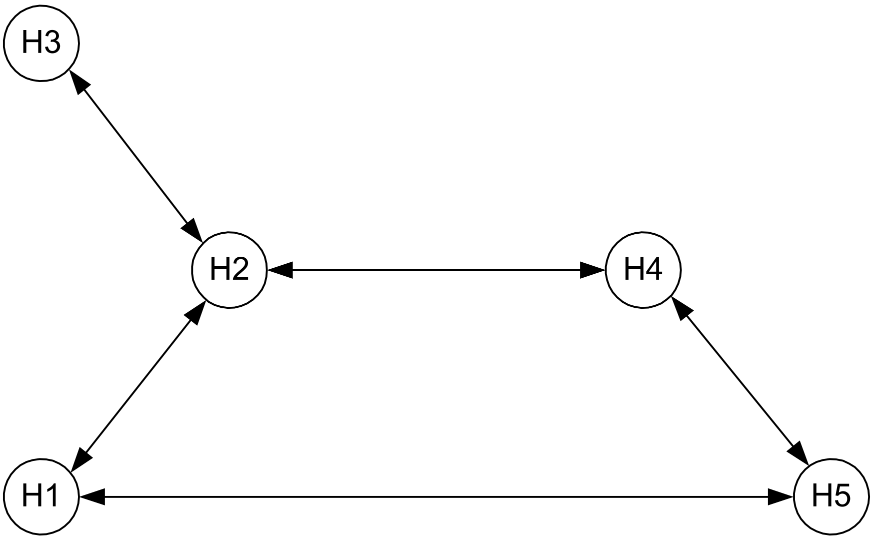


Figure 2-3: A simple topology

Table 2-1 illustrates the route information that the node H4 would store.

Dest	Next Hop	Metric	Seq. No	Install
H1	H2	2	S406_H1	T001_H5
H2	H2	1	S128_H2	T001_H5
H3	H2	2	S444_H3	T001_H5
H4	H4	1	S123_H4	T001_H5
H5	H5	1	S489_H5	T001_H5

Table 2-1: Routing table for H4 node in the DSDV protocol.

Table 2-1 illustrates the nodes only stores information about destination and next hop, and not about the entire route. As seen, the route from H4 to H3 goes through H2, which means that the metric is 2 (hops). The next node on the route from H4 to H3 is H2, and H4 will therefore forward packets for H3 to H2. Information concerning the next hop is stored in the Next Hop column.

The sequence numbers in the Seq. No column is used to compare routes. Routes with higher sequence numbers are considered more favorable. If the sequence number is the same the route with the lowest metric is preferred. The value in the Install column is used to help determine when stale routes should be deleted.

Each node in the network must periodically transmit its entire routing table to its neighbours. Missing transmissions can be used by neighbour nodes to detect changes (broken links) in the topology. Broken links may also be detected by communication hardware [Guoyou1]. When a broken link is detected it is assigned a metric value of infinity and the node that detected the broken link broadcasts an update packet, to inform others that the link is broken.

2.1.3 The Temporally-Ordered Routing Algorithm (TORA)

The Temporally-Ordered Routing Algorithm (TORA) protocol was developed by Vincent D. Park and M. Scott Corson [Corson1] in 1997. The protocol is an on-demand protocol that works by using a link-reversal algorithm such as the Gafni-Bertsekas (GB) algorithms [Perkins2]. The protocol is designed to minimize the reaction to topological changes and to discover multiple routes to a destination. Finding the shortest path is considered to be of less importance in this context.

Assumptions

A1. It is assumed that all transmitted packets are received correctly and in order of transmission.
A2. Bi-directional communication between nodes is assumed possible.
A3. It is assumed that a link-level protocol, which ensures that nodes always know their neighbours, is present.
A4. When a node transmits a packet it is broadcasted to all of its neighbours.

Mode of operation

TORA organizes the topology as a graph, were links (edges) between nodes can be in one of the following three states:

1. Undirected
2. Directed from node *i* to node *j*. In this case node *i* is said to be upstream from node *j*.
3. Directed from node *j* to node *i*. In this case node *i* is said to be downstream of node *j*.

The protocol uses a metric of *height of nodes* to direct the network. The method of operation is described as "*water flowing downhill towards a destination through a network of tubes that model the routing state of the real network. The tubes represent links between nodes, and the water in the tubes the packets flowing towards its destination*" [Broch1].

The protocol has three basic functions:

- Establishing routes
- Maintaining routes
- Deleting routes.

The establishment of routes essentially corresponds to assigning directions to links in an undirected network. To accomplish this task a query/reply process is used. The result of this process is a directed acyclic graph (DAG) [Corson1]. When a node needs to establish a route to a destination D, it broadcasts a QUERY packet with the destination. The packet propagates through the network until the destination or a node with a route to the destination is reached. The recipient broadcasts an UPDATE packet containing a value of the height that is a value that is assigned to the node. Every time a node receives the UPDATE packet it sets its height to one greater than its neighbours. This results in a set of directed links going from the sender of the QUERY, who has the highest height value, to the destination that has the lowest height value.

When a node discovers that a route is no longer usable it will increase its height to a local maximum with respect to its neighbours and transmit an UPDATE packet. This strategy minimizes the number of nodes that needs to be informed of the changes in the topology, since it correspond to a situation were water will flow back out of the node to the nodes that have been sending packets to it.

2.1.4 The Dynamic Source Route (DSR) Protocol

DSR was first introduced and described by David B. Johnson, David A. Maltz and Josh Broch in 1994 [Johnson1]. The protocol is specifically designed for use in multi-hop wireless ad hoc networks. The protocol does not require any existing network infrastructure or administration and is completely self-organizing and self-configuring. The protocol basically consists of the two mechanisms: Route Discovery and Route Maintenance, where the Route Discovery mechanism handles establishment of routes and the Route Maintenance mechanism keeps route information updated.

Assumptions

Some assumptions concerning the behavior of the nodes that participate in the ad hoc network are made. The most important assumptions are the following:

A1. All nodes that participate in the network are willing to participate fully in the protocols of the network.
A2. The diameter of the network are often small, e.g. in the interval of [5:10] nodes.
A3. Nodes can detect and discard corrupted packages.
A4. The speed at which nodes move is moderate with respect to packet transmission latency.
A5. Each node can be identified by a unique id by which it is recognized in the network.

Especially A1 is interesting since it builds on the trust and good will of other nodes in the network. The fact that Johnson et al emphasizes this as an assumption [Johnson1], indicates that they notice that the goodwill of other nodes might not always exist in practice.

Mode of operation

DSR operate on demand, which means that no data, such as route advertisement messages, is send periodically and therefore routing traffic caused by DSR can scale down and overhead packages can be avoided.
DSR is a source routing protocol, which means the entire route is known before a packet transmission is begun. DSR stores discovered routes in a Route Cache.

The two mechanisms: Route Discovery and Route Maintenance are described below.

Route Discovery

When a node S sends a packet to the destination D, it first searches its Route Cache for a suitable route to D. If no route from S to D exists in S's route cache, S initiates Route Discovery and sends out a ROUTE REQUEST message to find a route. The sending node is referred to as the initiator and the destination node as the target. The fields of the ROUTE REQUEST message are explained in Table 2-2.

Fields	Explanation
Initiator ID	The address of the initiator.
Target ID	The address of the target.
Unique Request ID	A unique ID that can identify the message.
Address List	A list of all addresses of intermediate nodes that the message passes before its destination. This is empty when the message is first send.
Hop Limit	The hop limit can be used to limit the number of nodes that the message is allowed to pass.
Network Interface List	If nodes have several network interfaces this information can be stored in this list.
Acknowledgment bit	There is an option of setting a bit so that the receiver returns an acknowledgement when a packet is received.

Table 2-2: Fields of the ROUTE REQUEST message. The Italic font are used to indicate fields used for the more advanced features of DSR.

The initiator initialize the Address List to an empty list and set the Initiator ID, the Target Id and the Unique Request Id in the ROUTE REQUEST message and then broadcasts the message. This causes the packet to be received by nodes within the wireless transmission range.

The initiator keeps a copy of the packet in a buffer, referred to as the send buffer. It timestamps the message so it can be examined later to determine if it should be send again. If no route is discovered within a specified time frame, the packet is dropped from the send buffer. Packets are also dropped from the send buffer if the buffer overruns.

When a node receives a ROUTE REQUEST message it examine the Target ID to determine if it is the target of the message. If the node is not the target it searches its own route cache for a route to the target. If a route is found it is returned. If not, the nodes own id is appended to the Address List and the ROUTE REQUEST is broadcasted. If a node subsequently receives two ROUTE REQUESTs with the same Request id, it is possible to specify that only the first should be handled and the subsequent discarded [Johnson2].

If the node is the target it returns a ROUTE REPLY message to the initiator. This ROUTE REPLY message includes the accumulated route from the ROUTE REQUEST message. The target searches its own Route Cache for a route to the initiator. The reason that the target node doesn't just reverse the found route and use it is that that would require bi-directional links. If a route is not found in the targets Route Cache, it performs a route discovery of its own and sends out a ROUTE REQUEST where it piggybacks the ROUTE REPLY for the initiator.

Route Maintenance

Since nodes move in and out of transmission range of other nodes and thereby creates and breaks routes, it is necessary to maintain the routes that are stored in the Route Cache. When a node receives a packet it is responsible for confirming that the packet reaches the next node on the route. Figure 2-4 that the mechanism works like a chain where each link has to make sure that the link in front of it is not broken. The figure also illustrates that node C might use another route to communicate to node A.

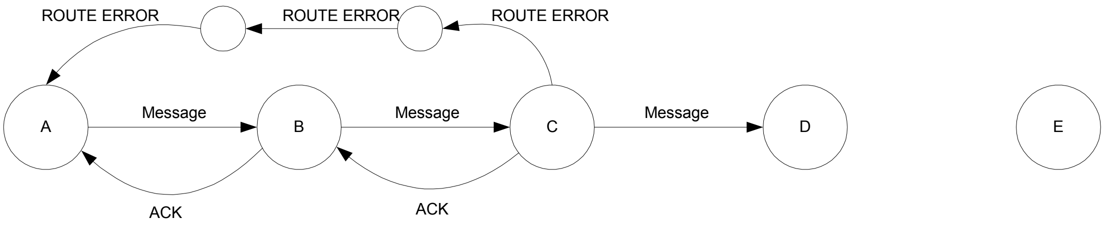


Figure 2-4: The acknowledgement mechanism works like a chain.

Acknowledgment can be performed either by using mechanisms in the underlying protocol such as link-level acknowledgment or passive acknowledgment. If none of these mechanisms are available, the transmitting node can set a bit in the packets header to request a specific DSR acknowledgment. If a node transmits a packet and does not receive an acknowledgment it tries to retransmit a fixed number of times. If no acknowledgement is received after the retransmissions, it returns a ROUTE ERROR message to the initiator of the packet. In this message the link that was broken is included. The initiator removes the route from its Route Cache and tries to transmit using another route from its Route Cache. If no route is available in the Route Cache a ROUTE REQUEST is transmitted in order to establish a new route.

Additional features in DSR

As explained in the above sections DSR has a quite simple mode of operation. However several additional features exist. This section gives an overview of these additional features. Some features such as *prevention of increased spreading of* ROUTE ERRORs and *storing of compromising information* exist, but are not covered here.

Caching of Overheard Route Information

Nodes can cache information about routes from packets that they overhear or forward. This mechanism is called snooping and is mostly used for snooping of routes. For example a node can cache the route that is returned in a ROUTE REPLY message when it forwards it. The use of route snooping can limit the amount of ROUTE REQUEST that are sends, since nodes can discover new routes this way.

Replying to Route Request Using Cached Routes

When a node receives a ROUTE REQUEST message for which it was not the destination it can attempt to find a route from its Route Cache instead of broadcasting the ROUTE REQUEST. If a route is found it is returned to the initiator. The node must however verify that the route that is being returned does not contain any duplicating nodes since this can lead to loops.

Avoiding Storms of Route Reply

When nodes are allowed to reply to ROUTE REQUEST messages with routes from their Route Cache the risk of ROUTE REPLY “storms” is present. “Storms” can occur when a node broadcast a ROUTE REQUEST and its neighbour nodes all has routes for the target in their cache. This will result in simultaneous ROUTE REPLYs from all neighbours that can cause congestion or packet collision. This can be avoided by letting the nodes delay ROUTE REPLYs for a random period. This delay effectively randomizes the time at which a node returns a ROUTE REPLY message.

Hop Limits on ROUTE REQUEST Messages

Table 2-2 shows the ROUTE REQUEST has a field that can be used to limit the number of hops that the packet may pass. This can be used to send non-propagating ROUTE REQUESTs and thereby query neighbour nodes to examine if one has a suitable route to the destination route in their Route Cache.

It is possible to use the hop limit to implement an expanding ring search. If the hop limit is increased by 1 every time a ROUTE REPLY is not received, as a result of a ROUTE REQUEST, the search for a suitable route will spread like a ring in the water. Johnson points out the risk that this expanding ring search could have the affect of increasing the average latency of Route Discovery [Johnson1].

Salvaging Packets

When a node forwards a packet, it might successively discover, through the use of Route Maintenance, that the route for the packet is broken. If the node has another route to the destination it can use it and thereby salvage the packet. If the packet is salvaged a ROUTE ERROR should be send to the original sender to report the link on the route that was broken.

Automatic Route Shortening

If a node overhears a packet that it eventually would receive it can return a “gratuitous” route reply to the original sender to inform the sender that a shorter route exists.

2.1.5 The Ad-Hoc On Demand Distance Vector (AODV) Protocol

The AODV protocol was presented in 1997 and is designed by Charlie E. Perkins and Elizabeth Royer, who also designed the DSDV protocol [Perkins1]. The protocol is a hybrid of the DSR protocol and the DSDV protocol. It uses a route discovery process much similar to the one used by DSR and makes use of hop-by-hop routing like DSDV. The primary objectives for the AODV algorithm are:

- To broadcast discovery packets only when necessary.
- To distinguish local connectivity management (neighbour nodes) from changes in the entire topology.
- To try to forward information concerning changes in local connectivity to neighbour nodes who are likely to need it.

Assumptions

A1. It is a requirement that the broadcast medium provides the means so nodes can detect neighbour nodes broadcast messages.

Mode of operation

As mentioned the route discovery process is much similar to the one used by the DSR protocol. AODV differs from DSR in the way that nodes do not store the entire route to a destination. The path maintenance mechanism used by AODV is quite similar to the one used by DSDV and therefore AODV will not be described further.

2.1.6 Comparison of Ad Hoc Routing Protocols

This section gives an introduction to some of the results of performance comparisons of ad hoc protocols that have been presented in literature [Broch1] [Johnson1]. The purpose of the section is to give the reader an idea of different protocols performance. Summarizing all results and conditions is not in the scope of this thesis.

Several performance comparisons of the protocols described in this chapter have been conducted over time, but none of these seems to have resulted in a unanimous recommendation of one protocol over the others and no standard has yet been adopted. One of the most comprehensive comparisons seems to be the one conducted by Josh Broch et al. in 1998 [Broch1]. DSR, DSDV, TORA and AODV are all compared using the same simulation environment (Ns-2) under similar conditions. The overall goal of the comparison was to measure the protocols ability to adapt to changes in the topology and still deliver packets. The protocols were evaluated using three metrics:

- Packet delivery ratio, the ratio between the number of packets that the application layer sends to the protocol and the number of packets received at the destination.
- Routing overhead, the total number of packets send during the simulation.

- Path optimality, the difference between the number of hops a packet took and the length of the shortest path.

The results show that DSR and AODV deliver over 95% of the packet regardless of the mobility rate of nodes used for the simulations. For DSR these results correspond well to other results presented in literature [Johnson1]. The abilities of TORA and DSDV to deliver packets depend on the movement patterns of nodes in the network; the more rapidly the nodes move the more packets be dropped.

It is concluded by several sources that DSR has the lowest routing overhead [Johnson1], [Broch1]. The fact that DSR packets contains a high number of bytes since the entire route is contained in the packet, is not given much significance, since the cost of transmitting a packet is much greater than the cost of adding some extra bytes [Broch1].

The presented results show that DSDV and DSR seem to route very close to the optimal routes. AODV and TORA have significantly worse results; 4 or more hops than the optimal route were measured for some packets.

2.1.7 Summary

In this section ad hoc networks have been introduced and different types of routing protocols for ad hoc networks, such as *proactive, reactive*, *source route based* and *hop-by-hop* based have been discussed. The four ad hoc routing protocols listed below have been described.

- DSDV, Destination-Sequenced Distance Vector protocol
- TORA, Temporally-Ordered Routing Algorithm
- DSR, The Dynamic Source Routing protocol
- AODV, Ad-hoc On-demand Distance Vector

The mode of operation and different assumptions for the protocols were covered. Finally some results from performance comparison of the four protocols have been pointed out. Based on the results from the performance analysis and the fact that DSR is based on source routing, which means that the whole route is known at the time of transmission, supports the decision to apply trust based routing to DSR. The research and investigations described in this section has lead to the fulfillment of the preliminary objective PRE-O 1.

2.2 Trust Management Systems

This section introduces trust management system in order to fulfill part of the preliminary objective PRE-O 3. Trust management systems are used to handle authorization issues in distributed systems. The basic idea of the trust management systems is that applications delegate authorization issues to a standard component, the trust management system. This prevents that each application has to implement its own authentication mechanism. The trust management system supplies languages to represent policies and credentials. This gives a unified mechanism so policies and credentials can be exchanged between different systems. Trust management systems only handles authorization issues, so it is required that applications that use trust management systems handle the appropriate integrity checks and signature validations.

Matt Blaze, who is one of the pioneers in the area of trust management, defines the following principles for trust management [Blaze1]:

- **Unified mechanism**: Policies, credentials and trust relationships are expressed as programs and existing programs are forced to treat these concepts separately.
- **Flexibility**: The system must be expressively rich enough to support complex trust relationships and at the same time it must be possible to express simple and standard policies succinctly and comprehensibly.
- **Locality of control**: Each node in a network can decide whether it will accept credentials presented by a second party or, alternatively on which third party it should rely for an appropriate "certificate".
- **Separation of mechanism from policy**: The method for validating credentials does not depend on the credentials themselves or the application that uses them.

The following sections will introduce three of the most prominent trust management systems. These systems have been selected because they are the most referenced in literature and because they are well documented.

2.2.1 PolicyMaker

PolicyMaker was developed by Matt Blaze et al. in 1995 [Blaze1]. The system was the first tool that embodied the four trust management principles described above. PolicyMaker is suitable for use together with services whose main goals are privacy, authentication and enabling of functionality.

PolicyMaker binds public keys to predicates. These predicates are used to describe actions that the keys are trusted to sign for. Unlike other systems, PolicyMaker does not deal with the identity of the user of the key, which makes the system ideal when anonymity is a requirement.

The PolicyMaker system can be thought of as a form of database that the application queries for answers of the questions of the type: "*May the key K perform action A according to the local policy LP* ". PolicyMaker takes as input a query of the form:

$\texttt{key}_1\texttt{, key}_2\texttt{,…,key}_n\texttt{ REQUEST ActionString}$

ActionString is an application specific message that corresponds to some kind of trusted action requested by one or more public keys. It is possible to add filters to the query so it only returns the ActionString if the filter predicate holds. PolicyMaker processes the queries based on trust information contained in assertions.
Assertions are of the form:

```
Source ASSERTS AuthorityStruct WHERE filter
```

Source indicates the source of the assertion, either a local policy (policy assertion) or a public key of a trusted third party (signed assertions). `AuthorityStruct` specifies the public key(s) to which the assertion applies.

2.2.2 KeyNote

The KeyNote trust management system is a direct successor of the PolicyMaker system. It is also developed by Matt Blaze and builds on the same ideas as PolicyMaker.

One of the main differences between PolicyMaker and Keynote is that KeyNote has a simpler syntax and semantics aimed specifically to build public-key infrastructure applications were PolicyMaker aimed to provide a framework for a wider range of applications [Blaze3].

Where the PolicyMaker uses queries, KeyNote instead uses an Action Environment. This Action Environment is passed from the application to the KeyNote system. The Action Environment is a triplet that consists of:

- The security policy
- A list of credentials
- The request.

As a result of passing this Action Environment, KeyNote returns an application specific string, which in the simplest case could be “Action authorized”.

Like other trust management systems KeyNote does note enforce the policies upon the system, it only provides advice to applications that call it.

2.2.3 REFEREE

The REFEREE (Rule-Controlled Environment For Evaluation of Rules and Everything Else) system is from 1997 [Yang]. The system differs from PolicyMaker and KeyNote in the way that it is designed to help making access decisions concerning web sites.

Like PolicyMaker and KeyNote it functions as an engine that can be queried for recommendations. The answer to a query can be true, false or unknown. Unknown means that the system was not able to make a decision about whether the requested action could be recommended using the policy that was in force. In such a case the calling application needs to determine which action should be taken. All trust decisions are based on policy control. This means that the system can be used to write policies about policies, policies about cryptographic keys, certificates or anything else.

2.2.4 Summary

This section has introduced the basic principles of trust management systems and has covered three of the most known trust management systems: PolicyMaker, KeyNote and REFEREE. The introduction has lead to fulfillment of part of the preliminary objective PRE-O 3. Trust management systems seek to answer the questions:

Is the request R signed by the key(s) K allowed under the policy P?

The idea behind trust management systems is to separate the development of the authorization mechanism from the development of the application. This is done by providing languages and API's that can be used to specify policies and credentials and a mechanism to evaluate requests based on the specified credentials and policies. Since the mechanism for verifying credentials does not depend on the format of the credentials themselves, different applications with different policies can share a single verification infrastructure.

2.3 Security In Ad-Hoc Networks

This section discusses some of the security issues that are related to wireless Ad-Hoc networks, and presents some of the proposed solutions to such issues.

The nature of wireless Ad-Hoc networks makes their security characteristics different from other kinds of networks. One of the major security obstacles is the absence of a fixed infrastructure, which makes it impossible to use existing trusted servers in the used security mechanisms. Ad hoc networks are vulnerable to security attacks on both physical and virtual levels. The devices (Laptops, PDA's, mobile phones, etc) are moved around which makes it hard to secure them physically. Furthermore, the use of wireless links makes the network vulnerable to link level attacks [Zhou]. On a virtual level many different kinds of attacks can be made on the routing protocols. As mentioned earlier in section 2.1, none of the protocols, DSDV, TORA, DSR and AODV, accommodates mechanisms to deal with any type of attacks or malicious behavior.

Sharp recognizes the following two main problems for obtaining security in a distributed system in general [Sharp].

1. It is difficult to protect physical connections, which means that one should assume that communication can be overheard, recorded for replaying or altered.

2. It is hard to determine whether or not the party you are communicating with really is who you think he is.

The two problems identified by Sharp actually cover several sub areas of security. Since security is such a complex field it is often divided into sub parts. Zhou et al identifies the following properties that need to be covered in order to obtain a secure ad hoc network [Zhou]:

Confidentiality: Ensures that certain information is protected from unauthorized disclosure.

Integrity: Guaranties that a message is never corrupted or modified.

Availability: Ensures that the network is functional despite of denial of service attacks and available when expected.

Authentication: Offers facilities for confirming that the node that one is communicating with is actually the party that one believes it to be.

Non-repudiation: Ensures that data has been sent or received by a particular party. Non-repudiation with proof of origin is used when the sender cannot deny that he was the sender of the data and Non-repudiation with proof of delivery prevents the receiver from denying that data was received.

The following sections introduce different approaches to obtain one or more of the security properties mentioned above.

2.3.1 Zhou et al Key Management Service

Zhou et al proposes a key management solution, which aims to ensure integrity, confidentiality and availability [Zhou]. The service adopts a public key infrastructure (PKI). It is assumed that *n* trusted servers are present in the network. Each of these servers has a public/private key pair and stores all public keys of nodes in the network. These servers know the public keys of other servers and can establish secure links among each other. The service has one public/private keypair *k* that is divided among the trusted servers. A message is given a digital signature in order to obtain integrity. The signature is given by the use of a threshold signature method [Desmedt]. The use of threshold signatures makes it possible to sign a message even though some servers in the network are compromised, and impossible for a compromised server to sign the message. Figure 2-5 illustrates the procedure

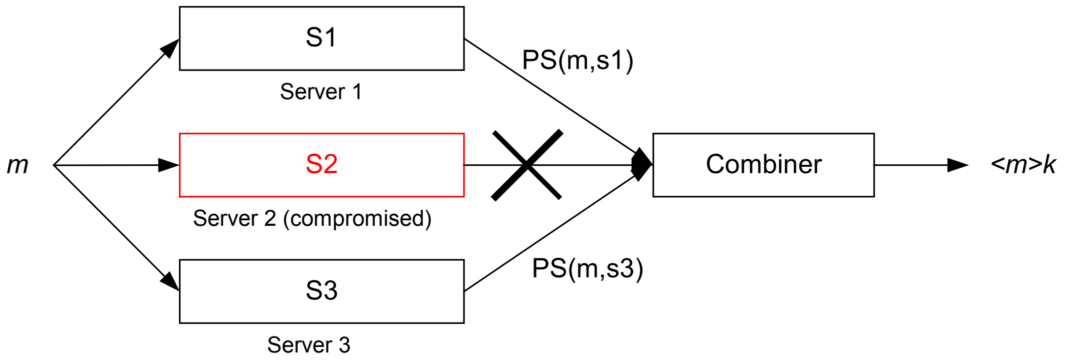


Figure 2-5: Threshold signature, even though server 2 is compromised it is still possible generate a signature.

Since the trusted servers do not gain any benefits, it its doubtful whether the proposed solution can be applied to a collaborative network.

2.3.2 The Security Aware Ad-Hoc Routing Protocol (SAR)

Yi et al proposes the SAR protocol to improve security in ad hoc networks [Yi]. Even though it is not stated directly in Yi's paper, the used examples and some of the areas that the paper cover indicates that the area of application assumes that a single authority is present in the network. The protocol embeds the metrics for the security level that an application wants to use into Route Request packets. If nodes receive a packet with a security metric or trust level that they cannot provide the packet is dropped and nodes that cannot provide the level of security will not be a part of the route. To incorporate trust levels in the ad hoc network Yi et al. proposes that existing organizational hierarchies is mirrored in the ad hoc network. Organizational

hierarchies may exist in many forms of ad hoc networks, e.g. search and rescue operations, major events organization etc. No proposals are presented on how to handle a situation were no organizational hierarchy exists. The protocol requires that nodes can be authenticated. To obtain authentication a simple shared secret is used to generate symmetric encryption/decryption key per trust level. Packets are encrypted on each trust level, which means that nodes on a different level cannot read the packets. The issue of how to distribute this secret in the first place is not treated.
The protocol is implemented as an augmentation to the AODV protocol and simulations were carried out using the Ns-2 [NS2] simulator. The results showed that even though the overhead per control message were higher, the performance were sustainable.

2.3.3 Entity Recognition

Seigneur et al recognizes that traditional authentication mechanisms such as Public Key Infrastructure (PKI) or Kerberos [Kohl] might not be in suitable for ubiquitous computing [Seigneur]. Traditional authentication schemes help to establishing the identity of an entity, by binding a secret key to an identity. However, this binding does however not give any information about how the entity is expected to act. Using entity recognition, an entity recognizes another entity but does not care about its identity. This has the advantage that entities can establish relationships without having pre-determined knowledge of each other. Seigneur et al proposes the, *A Peer Entity Recognition scheme* (APER), to be used to achieve entity recognition. In ad hoc networks the identity of a node might be less important, compared to the expected behavior of a node.

Entity Recognition

The Entity Recognition (ER) process is compared to the normal Authentication Process (AP). The comparison is illustrated in Figure 2-6.

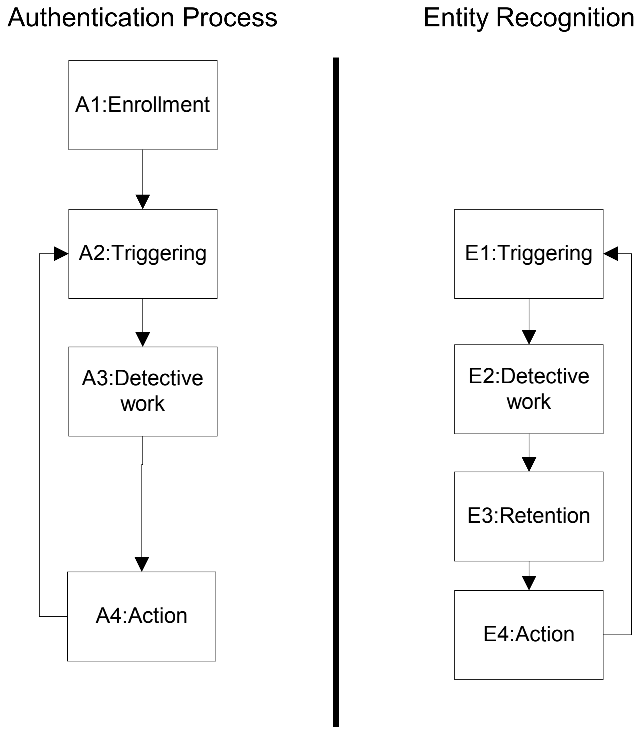


Figure 2-6: Comparison of Authentication Process and Entity Recognition

E1 In step E1 some kind of triggering takes place. Triggering can be self-triggering, which means that an entity takes initiative to recognize potential surrounding entities. Transmission of the DSR ROUTE REQUEST could be an example of self-triggering.

E2 In step E2 some detective work is done in order to see if the entity can be recognized. One way to do this recognition is to use the APER scheme, which is described in further details later.

E3 Step E3 is optional, since the recognition information does not need to be stored – for instance if the entity has been seen before.

E4 Step E4 is also optional, since no action has to be performed if the only objective of the process was to collect recognition information.

Figure 2-6 illustrates the first step of the normal Authentication Process A1: enrollment, that normally acquires an administrator, is unessesary for Entity Recognition.

A Peer Entity Recognition scheme (APER)

The APER scheme can be used to perform the step E2: Detective work of the ER scheme. APER requires that public key encryption is possible. APER uses two roles, the recognizer and the claimant. The basic approach is that the claimant occasionally broadcasts a digitally signed packet. At any time the recognizer can challenge the claimant if desired. When the recognizer sends a challenge to a claimant using the claimant's public key, the claimant needs its secret key to produce a correct response. If the response is correct, the recognizer can re-associate the public key with some context information such as the claimants network address or similar.
APER offers three levels of recognitions:

Level 1 requires that a claimant's signature be verified over a set of recently fresh claims.

Level 2 requires Level 1 to be fulfilled. Further, to ensure that the claim is "fresh" and not just copied from some another broadcast network, the claimant's claim must include the hashes of the last *n* claims. If the recognizer has one of these hashes the claim can be treated as "fresh".

Level 3 requires Level 2 to fulfilled, and further requires that the claimant can respond successfully to a challenge.

APER provides a strong recognition scheme when using level 3, and gives entities the possibility to establish relationships build on previous recognitions.

2.3.4 The Watchdog – Pathrater approach

Marti et al describes two techniques that can improve the throughput in an ad hoc network [Marti]. A Watchdog is used to monitor and identify malicious nodes in the network and a Pathrater to adjust nodes trust rating based on the number of packets they forward. It is a requirement for the watchdog technique to work, that the wireless interface supports promiscuous mode, which allows nodes to receive all packets that are transmitted within their range. The Watchdog monitors whether neighbour nodes forward packets as they were supposed to. If a packet is not forwarded the Pathrater will adjust the nodes trust rating in a negative way. If packets are forwarded, the node will update the trust rating of the forwarding node in a positive manner. The trust ratings are then used to determine which routes to use. It is important to notice that a node can only monitor if a neighbour forwards or not, not nodes that are two or more hops away. This means that other nodes have to detect a malicious node on the path and report it back to the destination. Malicious nodes are not punished for the misbehavior, and they still get their own packets forwarded while they at the same time is relieved of forwarding packets from others. This actually means that malicious nodes are rewarded for their behavior.

Marti et al identifies that the mechanism has some weaknesses that are listed below. The Watchdog cannot detect a misbehaving node in the presence of

- Ambiguous collision
- Receiver collision
- Limited transmission power
- False misbehavior
- Collusion
- Partial dropping

Marti et al. have implemented the Watchdog and Pathrater techniques in the DSR protocol. Simulations executed on the Ns-2 simulator, showed that the throughput in the network, when nodes are dropping packages, could be increased by up to 27 % by adopting the Watchdog and Pathrater techniques. The increase in throughput can however lead to a routing overhead of up to 24%.

2.3.5 The CONFIDANT protocol

The CONFIDANT protocol works as an extension to reactive source routing protocols like DSR [Buchegger1]. The basic idea of the protocol is that nodes that does not forward packets as they are supposed to, will be identified and expelled by the other nodes. Thereby, a disadvantage is combined with practicing malicious behavior. The protocol consists of four components:

- The Monitor
- The Trust Manager
- The Reputation System
- The Path Manager

The Monitor is used to monitor the behavior of neighbour nodes. It is possible to monitor, if packets are forwarded as supposed, unusually frequent route updates, etc. The monitor registers "'bad" behavior and notifies the reputation system so suitable actions can be taken. The Trust Manager sends out ALARM messages to warn friendly nodes of malicious nodes.

When an ALARM message is received the Trust Manager determines whether there is sufficient trust in the node that send the message, to avoid that innocent nodes are punished. ALARM messages are only communicated amongst friendly nodes. How to

establish friend relationships among nodes are still in the area of research but the CONFIDANT protocol adopts an approach known as, *The resurrecting Duckling* [Anderson]. It is usually described as the scenario, where ducklings emerge from their shell and identifies the first living creatures they see (dog, cat, duck, etc.) as their mother. It is most often used in computer science by establishing a friend relationship with the first entity that sends a secret key to the duckling [Anderson].
The Reputation System keeps a trust rating of nodes. This rating is changed when a node behaves malicious. Whether or not a nodes behavior is malicious is determined according to a threshold function. The ratings are only changed in a negative manner, which means that once a nodes trust is broken, there is no way to win it back. This is done since malicious behavior ideally is identified as an exception [Buchegger1].
The Path Manager ranks the routes according to their reputation and ensures that no malicious nodes are used in routes. It also handles request for routes from malicious nodes by simply ignoring them.

Buchegger et al. has implemented the CONFIDANT protocol on top of the DSR protocol and done simulations using the GloMoSim [Bajaj] simulator. Simulations showed that in some situations, DSR fortified with CONFIDANT lost less than 3% of the packets, were DSR alone lost up to 70% of all packets [Buchegger2]. Further simulations showed that the DSR fortified with CONFIDANT performed well with a fraction as high as 60% malicious nodes.

2.3.6 Nuglets

Buttyan et al. presents the idea of using so called nuglets as a virtual currency in ad hoc networks [Buttyan]. The nuglets are introduced to motivate nodes to corporate and provide services to each other. Two methods for implementing the nuglets are proposed: The Packet Purse Model (PPM) and the Packet Trade Model (PTM). In PPM the sending node attach some amount of nuglets to the packet being send. Each forwarding node then takes some amount of these nuglet in order to cover the cost of forwarding the packet. Two methods for performing this task in practice are proposed. One is to calculate a fixed charge *u* for each hop on the route and use a protection mechanism to ensure that each forwarding node only takes *u* of the nuglets. This approach requires that the sending node have knowledge of the total number of hops on the route. The other method is to have an auction where all neighbouring nodes bid on the price for forwarding the packet. The sending node then takes the lowest bid and forwards the packet. To implement this method Buttyan et al. propose to let nodes use an agent to perform the bidding on their behalf.
The PTM works a bit opposite; here the packet is traded for nuglets, so eventually the destination node will pay for the packets.
Several security mechanisms such as the presence of trusted and tamper proof hardware and a public key infrastructure is necessary in order to introduce nuglets. Further there are several unsolved issues such as how to get new nuglets, which can be a significant problem for nodes in the periphery of the network.

The main result of the simulations run by Buttyan et al. was that the introduction of nuglets did not decrease the performance of network significantly.

2.3.7 Trust based routing

In his master thesis from 2002, John Keane [Keane] developed and applied trust based routing DSR. The main idea behind trust based routing is to store information about the trust that one node has in other encountered nodes. These trust values are adjusted based on the nodes experiences, such as packet drops or acknowledgements receipts. This is different from the CONFIDANT approach because nodes only rely on their own observations. The source routes are evaluated based on some heuristic that uses the nodes trust value as criteria. Keane’s results showed that his implementation, in some situations, had a higher throughput than standard DSR. The results were however not ambiguous since they also showed that standard DSR outperformed the trust based routing in situations with a high number of malicious nodes. Another interesting aspect of Keane’s work is that the results clearly showed that malicious nodes had very low trust values, which indicated that they were identified as malicious nodes.

2.3.8 Summary

This section has covered some of the security issues, which has to be handled in order to obtain a secure ad hoc network. By examining these areas the preliminary objective PRE-O 2 was fulfilled. Several proposed solutions on how to achieve one or more of the security properties: Confidentiality, Integrity, Availability, Authentication and non-repudiation has been described.

Zhou et al key management system and The Security Aware Ad-Hoc Routing Protocol (SAR) is designed to function in environments where a single trusted authority is present and therefore they build on some strong assumptions that does not hold in collaborative ad hoc networks where such an authority is not present.

The key management system is designed to secure integrity, confidentiality and availability and builds on threshold encryption and distribution of keys among several trusted servers.

The SAR protocol incorporates trust levels in the network that reflects hierarchical structures of the domain where the ad hoc network is used. By using a shared secret on each level authentication and confidentiality is achieved.

The solution proposed by Marti et al. uses a Watchdog to identify misbehaving nodes and a Pathrater to manage nodes ratings. The CONFIDANT identifies misbehaving nodes, alarm friends about the misbehaving nodes and keeps a trust rating of nodes. Unlike Marti et al solution, the CONFIDANT protocol punishes misbehaving nodes by refusing to forward their packets. Simulations of both solutions showed an increase in throughput when malicious nodes were introduced in the network.

Buttyan et al introduce the use of nuglets as a virtual currency. Nodes use nuglets to pay other nodes for forwarding their packets. There are, however unsolved issues related to the case when a node runs out of nuglets. Simulations showed that introduction of nuglets did not decrease the networks performance significantly.

Keane introduced trust based routing and applied it to DSR and achieved increased throughput in some situations. By storing and maintaining trust values for nodes he was able to identify malicious nodes in the network.

2.4 Trust

Even though trust is widely used in our daily life, and by many people it is an extremely complex subject to work with. Many trust-based decisions are made on a subconscious level, and it is often difficult for people to determine why and if they trust one person and not another. Furthermore, one person's reasons for trusting somebody might differ from another persons. One of the reasons for its complexity is that it is difficult to define exactly what trust is. This is reflected by the many different definitions that exist in literature. Trust is also difficult to measure, which adds to the complexity of the subject. Further it seems that trust in some way is related to the risk that is associated with a given situation or action [Marsh]; one might trust a neighbour to look after ones goldfish, but not ones child.

This section introduces some of the most widely accepted definitions of trust and some of the properties of trust, together with two frameworks that can be used to express trust in a formalized manner.

2.4.1 Definitions of trust

As mentioned earlier, several people have defined trust in various ways. Some definitions seem to be more widely accepted than others. In the following some of the most recognized definitions of trust is introduced.

The perhaps most popular and recognized definition is stated by Morton Deutsch in 1962 (copied from [Marsh]):

> a) *The individual is confronted with an ambiguous path, a path that can lead to an event perceived to be beneficial (Va^+) or to an event perceived to be harmful (Va^-);*
> b) *He perceives that the occurrence of Va^+ or Va^- in contingent on the behavior of another person; and*
> c) *he perceives the strength of Va^- to be greater than the strength of Va^+.*
>
> *If he chooses to take an ambiguous path with such properties, I shall say he makes a trusting choice; if he chooses not to take the path, he makes a distrustful choice.*

The use of the word perception implies that trust is subjective [Marsh] and differs from person to another person. One might be bold to state that *trust lies in the eye of the beholder.* The definition also implies that some kind of analysis of the benefit of $Va^{+,}$ compared to the cost of Va^-, is performed in order to choose the path.

Another definition of trust comes from Diego Gambetta who has gathered thoughts from diverse areas such as economics and biology. In his work from 1990 he gives the following definition of trust [Gambetta]:

> *Trust (or, symmetrical distrust) is a particular level of subjective probability with which an agent assesses that another agent or group of agents will perform a particular action, both before he can monitor it (or independently of his capacity ever to be able to monitor it) and in a context in which it affects his own action.*

Gambetta expresses trust as a probability, which means that it can be given a value in the range from 0 to 1. Similar to the definition of Deutsch, Gambetta also notes the subjectivity of trust. Further the definition indicates that the ability to monitor whether the trusted action is performed is of importance. A difference between Gambetta's and Deutsch's definitions is that Deutsch uses trust in a person, where Gambetta includes trust in a group of agents.

2.4.2 Different categories of trust

Trust is subjective and depends on both agent and situation/action, which makes it difficult to specify general rules. To cope with this issue several attempts have been made to make some classification of trust.

In the exhaustive work "The meaning of trust" [Mcknight], that builds on analysis of more than sixty papers related to the area of trust, Mcknight et al. uses trust construct to categorize trust. The trust constructs relates to each other as antecedents and consequents, and facilitates scientific measurements and predictions. The trust constructs are:

- **System Trust**, which means the extent to which one believes that proper impersonal structures are in place to enable one to anticipate a successful future endeavor.

- **Dispositional Trust**, which is the extent to which one consistently trust in a wide variety of situations and in different persons.

- **Situational Trust**, which means the extent to which one intends to depend on a non-specific other party in a given situation.

- **Belief Formation Process**, which is the process where experience and information is used to create new trusting beliefs.

- **Trusting Belief**, which is the extent to which one believes another person in a situation, is able and willing to act in ones best interest. It differs from situational trust in the way that the person is known in this situation and there fore prior experiences can be used to determine the degree of trusting belief.

- **Trusting Intention**, which is the extent to which one party is willing to depend on another party in a given situation with a feeling of relative security, even though it is realized that it can have negative consequences.

- **Trusting Behavior** is the extent to which a person voluntarily depends on another person in a specific situation with a feeling of relative security, though it is realized that it can have negative consequences.

Figure 2-7 illustrates how the trust constructs are related.

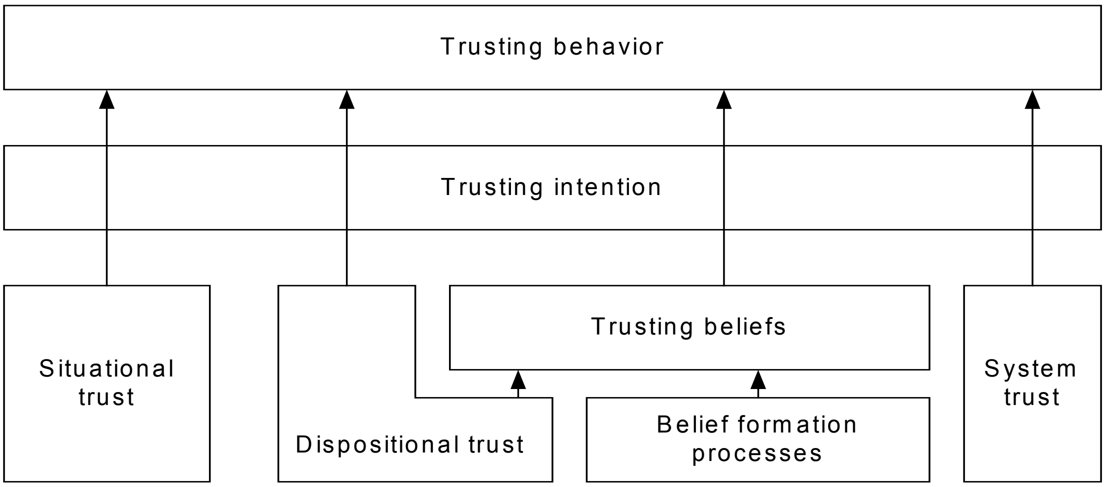


Figure 2-7: Relations between Mcknight et als. Trust Constructs

The construct above gives an insight in how one type of trust can lead to another. Stephen Marsh uses a somewhat similar, but simpler classification [Marsh], inspired by the work of Mcknight et al.

- **Basic trust**, which is derived from an agents past experiences in all prior situations. It represents how trustful an agent is. Basic trust corresponds to dispositional trust, from Mcknight et als categories.

- **General trust**, which represents the trust of one agent in another agent.

- **Situational trust**, which is the trust that one agent has in another agent in a given situation.

- **Importance and utility**, which expresses the utility an agent, gains from a situation.

The classification of trust leads to a more formal way of describing trust, which is covered in section 2.4.3.

There seems to be a general understanding that trust is not transitive, which can be illustrated by the following statement: ”If Ann trusts Bob and Bob trusts Cathy, then Ann trusts Cathy”. However the introduction of recommendations or similar can be used to construct transitive trust relations.

Trust depends on the situation that an agent is in. This is illustrated in Marsh’s framework by the introduction of situational trust.

There seems to be a general agreement in the literature that trust evolves over time, and can be adjusted in both positive and negative directions dependent on experiences. There is however no unambiguous agreement on how this evolution is expressed. It could be in a linear, exponential function or other kind of way. Further it is recognized that some sort of relation exists between risk and trust, since a decision to trust someone or something, is a decision laced with risk [Marsh].

2.4.3 Frameworks for Working with Trust

This section briefly introduces two frameworks that can be used to describe and work with trust in a formal manner.

Marsh's Framework

Marsh's framework makes it is possible to express trust values over time. The notation can be found in Table 2-3 and builds on the trust categories described in section 2.4.2.

Description	**Representation**	**Values**
Situation	$\alpha, \beta, \ldots$	
Agents	*a, b, c, …*	
Set of agents	*A*	
Societies of agents	$S_1, S_2, \ldots$; or $S_x \in A$	
Knowledge (e.g. *x* knows *y* at time t)	$K_x(y)^t$	True/False
Importance (e.g. of α to *x* at *t*)	$I_x(\alpha)^t$	[0,+1]
Utility (e.g., of α to *x* at *t*)	$U_x(\alpha)^t$	[-1,+1]
Basic Trust (e.g., of *x* at *t*)	T_x^t	[-1,+1]
General Trust (e.g., of *x* in *y* at *t*)	$T_x(y)^t$	[-1,+1]
Situational Trust (e.g. ., of *x* in *y* for α at *t*)	$T_x(y, \alpha)^t$	[-1,+1]

Table 2-3: Temporally indexed notation in Marshs framework

It is notable that trust can be assigned a crisp value in the interval [-1,+1]. By applying operators such as ∧ (AND), ∨ (OR) and ¬ (NEGATION) it is possible to express and derive complex rules. Since crisp values are used it is possible to do a numerical comparison and ordering. Several basic rules are introduced in Marshs framework from which others rules can be derived. It is considered out of the scope of this thesis to cover all of these, but to give the reader an idea and overview of how the rules looks some examples is given below in Figure 2-8 and Figure 2-9. The example in Figure 2-8 shows how the cooperation of y can lead to a higher trust value.
The more complex example in Figure 2-9 illustrates how a general trust value for agent *z* in agent *x* can be derived.

$$\text{Cooperate}(y)_x \Rightarrow T_x(y)^{t+1} \geq T_x(y)^t$$

Figure 2-8: The cooperation of y in a situation a leads

$$\neg K_z(x)^t \wedge K_z(y)^t \wedge K_y(x)^t \Rightarrow K_z(x)^{t+a} \wedge \left(T_z(x)^{t+a} \leq T_y(x)^t \;\right) \wedge \left(T_z(x)^{t+a} \leq T_z(y)^t\right)$$

Figure 2-9: Example showing how trust can be derived.

As Figure 2-9 illustrates *z*'s trust in *x* does not exceed the trust *z* had in *y*.
Marshes framework does not focus on a particular way to express the functions used to determine the crisp values of the different trust types.

Jonker and Treurs Framework

Jonker and Treurs framework focuses on the dynamics of trust based on experience [Jonker]. The framework introduces a basic notation and two types of functions: The trust evolution function and the trust update function. In the framework several possible properties of these functions are identified and expressed in a formal way.

The following types of initial trust are used:

1. **Initially trusting**
    a. Without prior trust influencing experiences, the agent has a unconditional trust of maximal trust value

    b. Without prior trust influencing experiences, the agent has a conditional trust value below the maximal trust value

2. **Initially distrusting**
    a. Without prior trust influencing experiences, the agent has a unconditional distrust of maximal distrust value
    b. Without prior trust influencing experiences, the agent has a conditional distrust above the minimum distrust value.

Jonker and Truer identifies the following six types of trust dynamics:

1. **Blindly positive**, which can be either:
    a. Always unconditional trust
    b. Definitive having trust: which means that after a certain number of positive experiences a state of unconditional trust is reached and remained in.
2. **Blindly negative**, which can be either:
    a. Always unconditional distrust
    b. Definitive losing trust: after a certain number of negative (sequenced) experiences a state of unconditional distrust is reached and remained in.
3. **Slow positive, fast negative dynamics**: It is hard to gain trust but easy to lose trust.
4. **Balanced slow**: The evolution in both positive and negative direction is slow.
5. **Balanced fast**: The evolution in both positive and negative direction is fast.
6. **Slow negative, fast positive**: It is easy to gain trust but hard to loose it.

The basis of the framework is the four sets shown in Table 2-4.

Set	Description	Possible values
E	A partially ordered set of experience classes for	{-,+}, [-1,1]
N	The set of natural numbers	0, 1, 2, ….
ES	The set E^N of experience sequences	{+, +,-,+,-} or {1, -1, 1, 1}
T	A partially ordered set of trust qualifications/trust values	{LOW, MEDIUM, HIGH} ,[-1,1].

Table 2-4: The four sets of the framework

The four sets are used to define the trust evolution function, the trust update function and several properties of the functions.

Trust Evolution Function
A trust evolution function, only uses past experiences to determine the trust value. The trust evolution function is defined as:

$$te: ES \times N \rightarrow T \qquad \text{Equation 2-1}$$

Jonker and Treur identify sixteen possible properties, such as *minimal* and *maximal initial trust*, of trust evolution functions. An interesting property is the *degree of memory based on window n* that expresses the number of experiences back in time that should be used in the calculation. This property is interesting since its value can lead to quite different results. Figure 2-10 illustrates a small example:

Trust evolution function	ES	$T_{n=8}$	$T_{n=4}$
$T = \frac{\sum_{1}^{n} ES}{n}$	[-1, -1, -1, -1, 1, 1, 1, 1]	0	-1

Figure 2-10: Using the *degree of memory based on window n* for trust evolution functions

As the example illustrates quite different result is achieved by using a window size of 8 instead of a window size of 4.

Trust Update Function
The trust update functions differ from the trust evolution function since it uses the current experience and the last calculated value of the trust to calculate the new trust value.
The trust update function is defined as:

$$tu: E \times T \rightarrow T \qquad \text{Equation 2-2}$$

A trust update function uses a prior determined value of trust and a new experience to calculate a new value for trust.

2.4.4 Summary

This section has introduced different definitions and key aspects of trust. To illustrate the complexity and versatility of trust, some of the most common definitions of trust have been discussed. Two categorizations of trust have also been covered. The categorizations lead to a better understanding of how the different types of trust are related. Furthermore some of the most important properties of trust have been identified. Two frameworks that make it possible to describe and work with trust in a formal way has been introduced.

The framework presented by Marsh is suitable for describing trust in different situation where the framework presented by Jonker and Treur focuses on describing the dynamics of trust based on experience. The dynamics of trust is described by introducing the two functions: the trust evolution function and the trust update function. Both frameworks propose to use the interval [-1,1] to represent trust values.

The completion of the investigation of trust and formal trust frameworks, has lead to the fulfillment of stated objective PRE-O 3.

2.5 Subjective evaluation of methods and techniques

Where the previous sections has sought to keep an objective perspective of the introduced areas, this section includes subjective evaluations of some of the covered areas and put into perspective if- and how they can be applied to develop trust based routing.

In section 2.1 the four protocols DSDV, TORA, DSR and AODV were described. Several assumptions for the protocols where summarizes. Marti et al [Marti] identifies that none of the routing protocols described in this chapter handle security issues. All assumes that no nodes are malicious and are willing to participate in the routing protocol.

Section 2.1.6 dealt with performance evaluations done for some of the protocols. The results of the evaluations indicated that DSR and AODV both delivered a high percentage of the send packets. One of the main differences between DSR and AODV is that DSR uses source routing and stores the entire routes, where AODV only stores information about the next hop on the route and used hop-by-hop routing. It would be quite a challenge to apply trust based routing to AODV since the only information that is available to build some sort of trust based decision on, is the next node on the route.

Since DSR is a source route protocol and therefore stores the entire routes it is possible to make a much better trust based evaluation of the route. The above observations have contributed to the decision on applying trust based routing to the DSR protocol.

In section 2.2 trust management systems were introduced. Trust management systems can be used to handle authorization and access issues. However, this is not really applicable to the type of ad hoc routing that is the foundation for this thesis and therefore none of the covered trust management systems is incorporated in the designed protocol.

In section 2.3 several different approaches for achieving different levels of protection against malicious behavior in ad hoc networks were introduced. Solutions, such as the Security Aware Ad-Hoc Routing Protocol (SAR) and Zhou et als key management system, offers encryption and a relatively high level of security, but also requires trusted servers to achieve this level of security. These requirements make them less applicable to the ad hoc routing used for this thesis.

Investigations showed that by using the APER protocol it is possible to establish relationships with nodes and recognize encountered nodes. By using APER it is possible to achieve authentication in ad hoc networks.

Other protocols such as CONFIDANT and the Pathrater – Watchdog mechanism monitor nodes to detect packet drops. As mentioned in section 2.3.5 Buchegger et al achieved good results with DSR fortified with CONFIDANT. However the mechanism of using ALARM messages to adjust the rating of paths seems rather vulnerable to misuse, since friendly nodes send out ALARM messages, but it is not really clear what happens if a friend acts malicious. Further, the process of establishing friends is still under investigation. Both methods have problems with detecting collusion because they rely on other nodes to report malicious behavior. How collusion can be handled is not treated which is somewhat a deny of a problem.

Keane achieved improvements on throughput in some situations by applying trust based routing to DSR. Further he showed that storing and updating trust information could be used to identify malicious nodes. Keane's methods differs considerable from the Pathrater – Watchdog and CONFIDANT because he does not rely others to report and observe malicious behavior, which eliminates problems with collusion and establishment of friends. However, Keane's results where somewhat ambiguous since standard DSR could outperform his implementation even though the percentage of malicious nodes where high.

In section 2.4 the two frameworks by respectively Marsh and Jonker and Treur were briefly presented. Since trust can be a rather informal area, it is found important to use some formal ways to describe the trust processes that is going to be used in this work. Marsh's framework is found well suited to express the different types of trust that are used in certain situations. Jonker and Treurs framework is aimed at describing the dynamics of trust and is found well suited to describe the evolution of trust.

Most of the results that are summarized from other sources [Johnson1], [Broch1], [Buchegger2] are achieved by doing simulations on either the Ns-2 simulator [NS2] or the GloMoSim simulator [Glom].
GLOMOSIM is implemented in the language Parsec [Parsec], which is a C based language where Ns-2 is based on C++. It is chosen to use the Ns-2 simulator because it has a C++ implementation of DSR.

3 Analysis

To meet the main objective M-O 1, the DSR protocol is analyzed to identify different situations where malicious nodes may exploit vulnerabilities in the protocol. Possible solutions on how to handle these exploitable situations are presented. Furthermore some of the extensions that are made to the DSR protocol are investigated.

There are several vulnerabilities in DSR that can be exploited by malicious nodes. In order to fortify the DSR protocol it is necessary to find ways to estimate whether a node is malicious or not. During the analysis situations were one nodes trust in others can be established or updated are identified. Some vulnerabilities cannot be handled by introducing trust and therefore the analysis also identifies situations were trust is found insufficient to make a useful defend. The identified issues are prioritized by assigning high priorities to issues where trust can be applied.

3.1 Analysis of DSR

This section presents an analysis of the different events that can take place during communication within the DSR protocol, the possible outcome and possible reactions to the outcome.
The relevant events can be divided into the following three categories:

- Sending of packets – the case where a node is the source of the packet.
- Receiving of packets – the case where the node is the final destination of the packet.
- Forwarding of packets – the case where a node is a hop on the route to the final destination.

3.1.1 Sending of packets

The following issues concerning the sending of packets are identified:

Using ring search to discover routes

When a node has to send a package it queries its route cache for a route. If no route is found it broadcasts a ROUTE REQUEST. It is a possibility to perform a ring search as described in section 2.1.4. Since ring search will limit the number of routes that are actually discovered and thereby increase the risk that all returned routes to a node include malicious nodes this might not be a good approach to use.

Selection of the "best" route

DSR selects routes from the cache based on the number of hops on the route, favoring short routes. This route selection strategy must instead be based on trust heuristics. The area of trust based route selection strategies are investigated further in section 4.1.3. An important observation is the fact that once DSR has a route to a destination it will use this route as long as it believes that it is working, meaning until it receives a ROUTE ERROR. This represents a serious problem in the case where only one route that contains a malicious node is available. It requires changes to DSR to poll for new routes if the routes available have to low a trust value and the consequences, such as increased ROUTE REQUESTs is somewhat unclear. Therefore introducing new mechanisms to discover new routes in the case where only one route containing a potential malicious node is available is considered out of scope of this assignment.

The use of DSR acknowledgement mechanism

As illustrated in Table 2-2 there is an option that allows the sender to set an acknowledgement bit in the DSR packet header in order to request an acknowledgement of the packet. If malicious nodes flip this bit it can prevent the receiver from returning acknowledgements or cause the receiver to send acknowledgements that was not requested. Since it is difficult to identify positive node behavior that could lead to an increase in trust without the use of acknowledgements there must be some mechanism for requiring acknowledgements. Therefore some extra fields are added to the packet header for this purpose. The area of acknowledgements strategies is covered in further details in section 4.1.5.

3.1.2 Receiving packets

This section describes and evaluates issues that can occur when a node receive a packet.

Forming trust relationships and updating trust

When a packet is received, the reversed route is stored in the cache. The route might contain unknown nodes which means that an initial trust relationship with these nodes must be established. When a packet is received it also means that all nodes on the route actually forwarded the package correct. This knowledge is used to update trust values for the nodes on the route. There is a chance that the source of the route is malicious and does not forward packages. Therefore the trust value for the source of the route is not updated. Trust updating is described in further details in section 4.1.2 and trust formation is described in section 4.1.1.

ROUTE ERRORs

When a ROUTE ERROR is received it means that a link is broken which causes DSR to select a new route and truncate the broken route from the route cache. It could be beneficial to increase trust for the nodes that forwarded the ROUTE ERROR and decrease trust for the node that caused the error. There is also a risk that a malicious node uses ROUTE ERRORs for framing, for instance by issuing false ROUTE ERRORs.

A malicious node forwarding a ROUTE ERROR might alter it so it looks like a different link on the route is broken or it might issue a false ROUTE ERROR. It is difficult to determine if the link between two nodes is broken without relying on one of them. It would involve cooperation of nodes in the neighbourhood to monitor the link. To discard a ROUTE ERROR message and still use the route would be useless since it would not be possible to decide whether the link was really broken or if a malicious node had forged the ROUTE ERROR. Therefore the only suitable solution is to use another route to the destination. If the ROUTE ERROR comes from a node, with a low trust value and the route cache contains a recently used route that the link was part of the trust value of the node could be adjusted in a negative manner. This would however mean that nodes that act according to the DSR protocol could be punished and for this reason it is decided to let DSR handle ROUTE ERRORs.

ROUTE REPLYs

It is possible that malicious nodes return fictive non-usable routes in ROUTE REPLYS. One way to deal with this is to evaluate the received route based on the trust in the sender and not trust, route reply’s from nodes with low trust. This evaluation would however apply to all returned routes and cause extra computations and it would not solve the problem where a malicious node forged the reply so it seemed to be initiated by some other node. To avoid the risk that nodes alter routes, the entire route could be signed and the signature included in the payload as well as in the packet header. This would of course increase the size of the payload and it would require that cryptographic methods were available. Another method for verifying the route is to let all nodes create a hash value based on the address of the node that forwarded the packet and thereby successive confirming each step of the route.

As pointed out in section 2.1.4 it is optional to reply to route replies from the route cache. By replying with a route from the route cache the number of forwarded route replies are limited which decreases the overhead on the network. On the other hand it might prevent new routes from being discovered. If only one route that includes a malicious node is contained in the cache this route is returned. This might represent a problem because it can cause bad routes to be distributed.

Another issue is that the cache represent some a learned part of the topology. Some of the information stored in the cache might not be valid because nodes can have moved out of range, turned their power of, etc. This problem is known as cache staleness. Hu et al points out that, replying from the route cache and at the same time allowing route snooping might result in stale routes circulating the network indefinitely [Hu]. Even though Hu et al proposes a solution, to this problem, it is not investigated further here, but simply noticed that the problem exists.

When a ROUTE REPLY is received, the route might contain previously un-encountered nodes. This means that the node should establish an initial trust relationship with these nodes. The new nodes are therefore given a trust value based on a trust formation strategy.

3.1.3 Forwarding of packets

Issues of forwarding packets are mostly related to behavior of malicious nodes.
The following issues concerning the forwarding of packets are identified:

Drop of forwarding packet

All forwarding packets can be dropped. There is nothing that can be done to prevent that a nodes drops a packet. The Pathrater – Watchdog technique and the CONFIDANT protocol that was described in section 2.3.4 and 2.3.5, uses a monitor technique to detect when a node is not forwarded packages. These techniques do however not adequately solve the problems of framing and collision among malicious nodes and therefore it is decided to use another approach where nodes only base decision on their own experiences and not on events that other nodes inform them of.

Tampering with ROUTE REQUESTs

When forwarding a ROUTE REQUEST message a malicious node can choose not to add its own address to the address list, which will probably cause the route to be useless. There is no way for the initiator of the ROUTE REQUEST to directly detect that the route has passed through a node that has not added it self. A solution to verify that the address list is not defective has been proposed in the section ROUTE REPLYs were the reception of ROUTE REPLYs was discussed.

Tampering with the address list

When forwarding a packet, a malicious node can choose to remove prior encountered nodes on the route from the address list or insert other nodes. This would most likely result in a useless route. This would especially be critical if the packet was a ROUTE REQUEST, since it eventually would result in a useless route. In order to avoid that nodes remove other nodes from the address list it would require each node to sign its own address in the list.

Snooping of source routes

As described in section 2.1.4, DSR has the possibility of snooping routes, which means that when a node forwards or overhear a packet it can snoop the route and cache it for later use. If a snooped route contains unknown nodes initial trust relationships must be established. The use of route snooping will decrease the amount of ROUTE REQUEST packages that are send, but in some situations it might limit the number of routes that are discovered to a destination because nodes that has a route to a destination will not issue new ROUTE REQUESTs to discover new routes to that destination. Figure 3-1 illustrates how node C might not discover some routes. However it is likely that the route can be snooped from some other route.

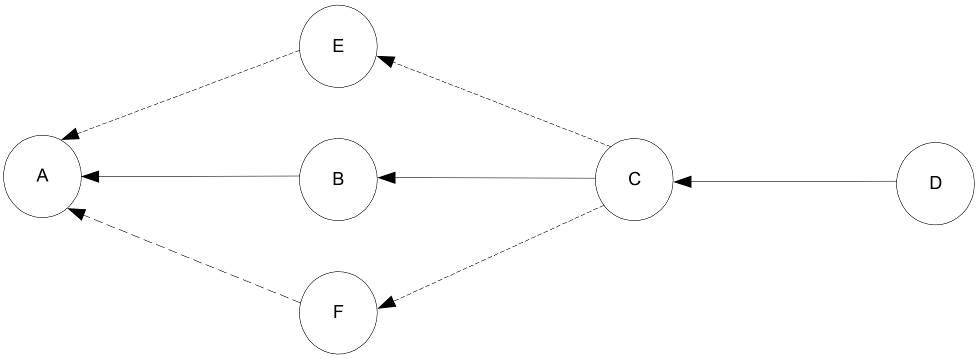


Figure 3-1: If node C snoops the route from D to A, it might never discover the two routes going through E and F.

Due to the increased overhead that disallowing snooping will cause, it is chosen to allow route snooping.

Only forwarding the first route request received

As described in section 2.1.4 it is possible to let nodes reply to the first received route request and discard subsequently received route request with the same unique id. As Figure 3-2 illustrates this can lead to potential routes being undiscovered. The figure illustrates a situation where node D has broadcasted a ROUTE REQUEST. Node C has propagated this to nodes E, B and F. Node B is malicious and its ROUTE REQUEST is first received at node A. This means that the routes going through E and F will not be discovered and D will be stuck with a route with a malicious node.

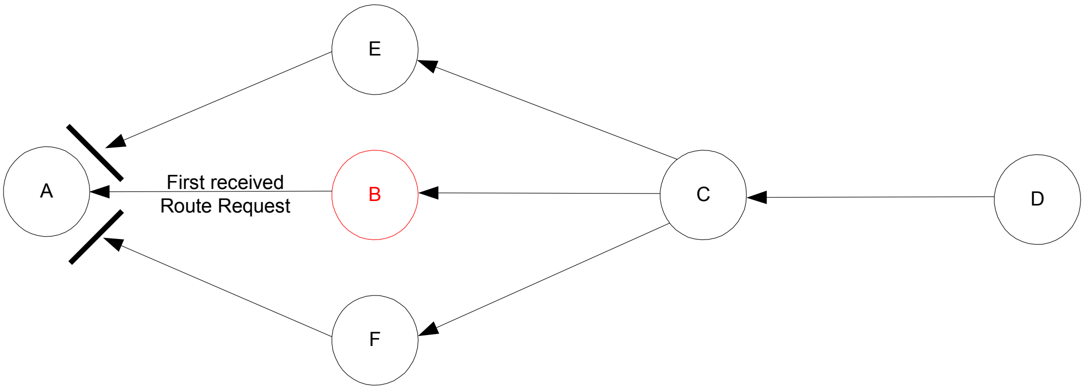


Figure 3-2: Only forwarding the first received ROUTE REQUEST can result in undiscovered routes.

It is expected that it is beneficial to let nodes reply to all route requests because it can increase the number of discovered routes and thereby decrease the probability of only discovering routes with malicious nodes.

3.2 Extensions to DSR

The above analysis revealed that the use of the DSR acknowledgement mechanism might not be suitable, since it is easy for malicious nodes to flip the acknowledgement bit and thereby tangle up the protocol. The only possibility for evolving trust relationships has this far been on receipt of packages. This is however insufficient, since a node might never receive packages that was forwarded by a certain route and therefore not be able to identify bad routes. In order to create a mechanism that can be used to base trust evolution on extra fields should be added so acknowledgement could be requested.

The analysis also revealed that the route selection criteria used by DSR should be altered to trust based instead of using a shortest path heuristic.

3.3 Assumptions made about malicious nodes

In order to have a simple setup as a starting point, the following assumptions will be made about the malicious nodes in the scenarios.

- Malicious nodes will not forward any packages for other nodes. This behavior is the most common one used by other people [Buchegger1] doing research in the same area and therefore it is a natural choice if the results from the simulations should be comparable.

- Malicious nodes will not return acknowledgements. This assumption is made to start with a simple scenario. By making this assumption it will be reasonable to increase trust for all nodes on a route that returned an acknowledgement. If malicious nodes also returned acknowledgements the source of an acknowledgement should have its trust value increased. However, it will also mean that nodes that successfully forward packets to malicious nodes will have their trust values decreased. Unfortunately time did not allow evaluation the impact of this assumption had compared to letting malicious nodes return acknowledgements.

The malicious behavior specified above, means that malicious nodes does not forward route replies, but they do however reply to route replies, meaning that if a malicious node has a route to a destination it will return it. If the returned route includes the malicious node a route containing a malicious node is returned. This makes impossible to avoid routes that do not contain malicious nodes.

Furthermore, one should keep in mind, that good nodes can also drop packages unintended due to queue overruns, low battery etc, which means that a good node unintended can exercise malicious behavior.

3.4 Attacks that are not covered by the analysis

There are some attacks against the DSR protocol that are not analyzed in this thesis. This does however not mean that they does not represent possible problems, but simply means that there were considered out of the scope of this project. Some of these attacks are:

- Denial of service attack where a node is bombarded with traffic.

- Traffic analysis, which is not a real attack. Introducing trust based routing might cause traffic analysis to become a problem since nodes that forward packages will be preferred. This could be exploited by forwarding all packages for a node and thereby achieving a high trust value. A node with a high trust value would most likely be exposed to a high traffic flow that could be analyzed.

3.5 Prioritization and general assumptions

A prioritization of the solutions is done in order to restrict the design and implementations. The solutions are prioritized using an estimate of the importance to the overall goals, of fortifying DSR against the presence of nodes that does not forward packages, by introducing trust based routing, and the expected complexity (and the involved expenditure of time) of implementing the solution.

It is chosen to design and implement the following:

- Trust formation mechanisms.
- Methods for updating trust based on experiences
- Methods to handle receipt and time outs of acknowledgements
- Trust based route selection

A solution that implements these areas is found to be sufficient to fortify DSR against malicious behavior that express it self as nodes dropping packets that they were supposed to forward. This means that all issues concerning tampering with packet are not handled as well as issues concerning authentication and confidentiality. Therefore the solution will build on the assumptions that these issues are handled. As section 2.3 illustrated solutions to handle these issues exists and therefore the assumptions are considered reasonable.

3.6 Summary

This chapter has presented an analysis of the DSR protocol where vulnerabilities has been identified and discussed. Solutions to handle these vulnerabilities have been proposed. It has been decided to design a solution that will implement the following:

- Nodes will store trust values of each encountered node that express the nodes trust. These values will be adjusted based on the experiences that a node has with other nodes.

- In order to require acknowledgements for received packages an extension to the existing DSR header will be implemented.

- When nodes receive acknowledgements or data packets they will update trust values for the nodes on the route, based on some trust updating policy. Nodes that are encountered for the first time, will have an initial trust value assigned based on some trust formation strategy.

- If a requested acknowledgement is not received within some timeframe the nodes on the used route should have their trust values decreased.

- Route selection will be based on some strategy that uses the trust in the nodes on the route to conduct an evaluation of the entire routes trust value.

- The extension will only seek to deal with malicious behavior that express it self as nodes dropping packets that they were supposed to forward.

How these task are handled is described in chapter 4. The decision to implement a solution covering the above tasks have also meant that issues concerning identification and tampering with packets is not treated further, but is assumed to be dealt with.

4 Design

This section presents the overall design of the modules and strategies that are going to be applied to the DSR protocol in order to incorporate trust formation and trust updating which are necessary to meet the main objective M-O 2. An implementation of the DSR protocol exist in the Ns-2 simulator that is used for the simulations and therefore parts of the design will have to be compatible with the existing code. The chapter also covers design of the different components that are used to incorporate trust in the DSR protocol. Furthermore, the existing code is analyzed and a UML class diagram is derived. Finally a description of how the design can be integrated with the existing code is given. The notations from section 2.4 is used to describe the types of trust that are used in the different situations. Since the design is object-oriented, object-oriented notations will be used, which means that functions will be referred to as *methods* even though there is a general agreement among C++ programmers that the word *function* is used. To keep consistency, method will also be used in chapter 5 that deals with the implementation.

4.1 Identification of components

From the analysis conducted in section 3.1 some major trust related processes are revealed:

- Initializing trust relationships
- Updating trust values
- Route selection based on trust

These processes provides the foundation for the trust based routing protocol that is going to be designed and will be implemented in one or more modules. Based on the analysis, components to manage the following tasks are needed:

- A structure to represent the trust that one node has in another node. It must also be possible to store the experiences that a node have had with other nodes since this is required in order to apply a trust evolution function as described in section 2.4.3.

- A component that determine the degree of trust to unknown nodes when they are encountered for the first time.

- A component that implements methods to update trust for known nodes.

- A component that evaluates routes according to the trust values of nodes on the route.

- A component that manages incoming and outgoing acknowledgements.

Further more, a component that can store and manage access to trust values is also needed. Figure 4-1 illustrates the overall design.

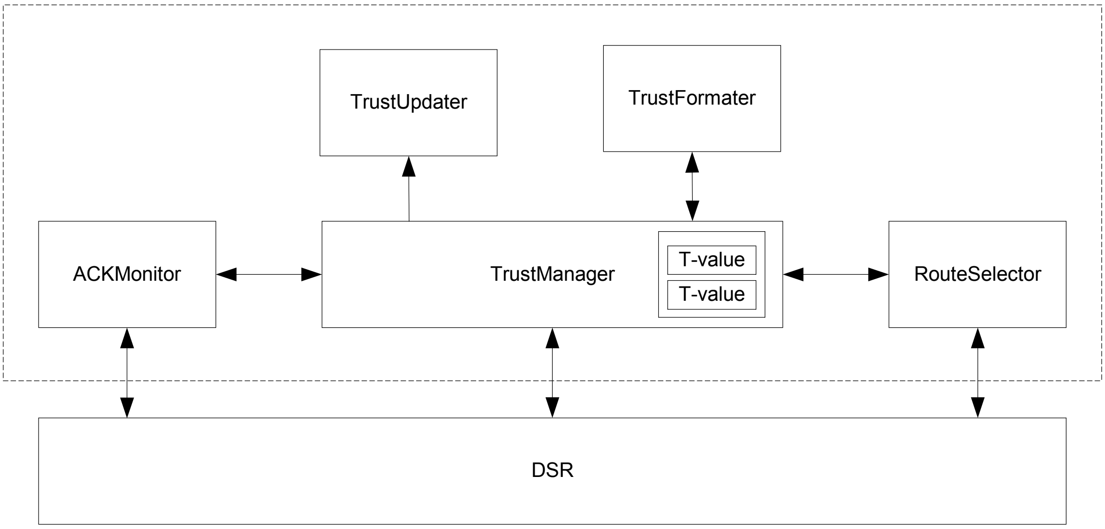


Figure 4-1: Overall architecture of the trust based extension

4.1.1 Trust Formation

The trust formation component implements methods to assign trust values to nodes when they are first encountered. Nodes will be encountered from ROUTE REPLYs or when routes are snooped. When the first routes are discovered all nodes on the route will be unknown and therefore a default value trust value for unknown nodes needs to be determined. The value of this parameter is quite important because it determines how close the node is to achieve maximal trust. In an environment with many malicious nodes it would be expected that it would be best to assign a low value trust value, meaning that the node would be initially distrusting and therefore having a low basic trust. In a situation where the route does not contain known nodes, the initial trust value will be the nodes basic trust. Because of the expected importance of this parameter it is estimated from simulations as described in section 6.5.1. If a route contains known nodes the trust value of these nodes is used to base the assignment of initial trust. Since the route is later going to be evaluated based on the trust value of each node, a strategy were the lowest trust value of the known nodes are assigned to the unknown nodes. The argument for choosing this strategy is, that the route is not stronger than its weakest link, which is the node with the lowest trust. However, several other strategies can be thought of for this purpose.
This strategy is described using Marsh's notation from section 2.4.3 on Equation 4-1.

$$\forall y \, \neg K_x (y_1, y_2 \ldots y_n) \Rightarrow (T_x(y_1, y_2 \ldots y_n) = T_x)$$

Equation 4-1

$$\exists y \, K_x(y) \Rightarrow T_x(\neg K(y_1, y_2 \ldots y_n)) = \mathrm{Min}(T_x(K(y_1, y_2 \ldots y_n)))$$

Where $y_1, y_2 \ldots y_n \in$ Route

4.1.2 Trust updating

The trust updating component implements the functions for updating trust. As mentioned earlier trust is subjective and depends on a given nodes experience in a given situation. This means that it is not reasonable to construct a general method for updating trust values that will be applicable to all applications in all domains. The function designed here is aims to function in domains with several malicious nodes.

A function for updating trust can depend on several parameters. In the list below some of the possible parameters are listed.

- Previous trust values.
- Lowest/Highest trust value ever assigned.
- Nr of positive/negative experiences in the past.
- The situation/value of an experience.

Ideally the behavior of the function should depend on the expected trust dynamics in a given situation. Since the trust values are used to base decision on which route to choose, only one function is used. Jonker and Treur propose the following trust update function [Jonker]:

$$f_d(ev,tv) = d * tv + (1-d) * ev$$ Equation 4-2

tv: The existing trust value
Where ev: The experience
d: A constant used to express the inflation of trust

Based on the observation that Jonker and Treur propose the intervals [-1,1], that interval is used for both experiences and trust values. The experience set will consist of three experiences that correspond to: Acknowledgement received ok, Acknowledgement timed out and Data packet received. An acknowledgement indicates that all nodes on the route could be trusted and therefore a value of 1 (maximum trust) will be assigned to this experience. The opposite counts when an acknowledgement is not received within the timeframe. This means that it must be assumed that the packet never reached its destination. Since the cause cannot be determined a value of –1 is assigned to this experience. The final experience is receiving a data packet, which means that nodes on the route have forwarded the package. This is not considered as powerful as an acknowledgement, because an acknowledgement is a response and confirmation of the trust that a node put in other nodes, where the receipt of data packet can be seen as a recommendation from the source to the destination. Therefore this experience will be given a value of 0.7. Figure 4-2 illustrate the impact that the inflation constant *d* has on the behavior of the function.

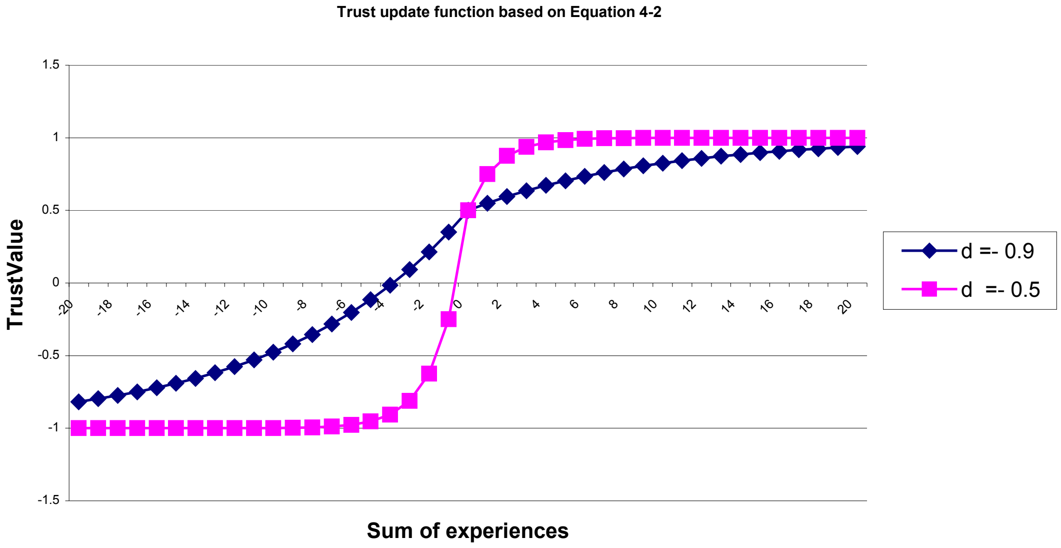


Figure 4-2: Trust update function based

The graph illustrates that a high *d* value will lead to a faster trust evolution towards maximum (or minimum) trust value. Only full positive experiences with a value of 1 and pure negative experiences with a value of -1 are used and the initial trust is set to 0.5. With a *d* value of -0.5 maximum trust will be placed in node after only a few experiences. This corresponds to trust evolving as balanced fast. With a *d* value of -0.9 trust will evolve as balanced slow. It is also possible to use one value for *d* when for positive experiences and another for negative experiences and thereby let trust evolve faster in one direction than the other. Based on the fact that a good node can also drop packages unintended d will be given the value -0.9. This means that good nodes that accidentally drop packages will not loose trust to fast. At the same time it means that a node with a high trust value that causes a negative experience will have its trust value lowered in a perceptible way.

4.1.3 Route selection

The main task of the route selection component is to evaluate routes based on the trust value of the nodes that constitute the route and selects a route based on this evaluation. The routes are evaluated and the route with the highest rating should be used. This means that the best route will be considered as the one that has the highest trust rating. A good route is considered to be a route that does not contain malicious nodes. To decide whether one route that results in a packet being delivered is better than another that achieves the same is difficult. Here metrics such as latency could be used. It can be concluded that a route that contains a malicious node is not good because it will always result in a packet drop. As the coming discussions of route strategies will illustrate, determining the best route can actually lead to a good route being discarded and a route containing a malicious node being chosen. Defining a route selection strategy is not an unambiguous task. Nodes are grouped as one because they are on the same route, but actually they do not have anything in common that, from a sociological point of view, can substantiate the grouping. This makes it quite difficult to argue for one routing strategy over another and therefore several route selection strategies will be proposed and used in the simulations.

Common for all routing strategies is that must not take the destination of the packet into account when the rating of the route is calculated, because the destination might be identified as a malicious node and therefore have a low trust value. This is necessary because the traffic is generated randomly for the simulations, and therefore malicious nodes will also be the destination of packets. Whether or not a node would actually send packets to a node that was identified as malicious is not treated here.

Furthermore, all strategies return a maximum rating if the route only consists of two nodes, since that means that the destination is a neighbour. If a maximum rating is returned, the route is used without examining further routes. This is actually a performance improvement compared to the implemented strategy used by standard DSR in Ns-2, where all routes to a destination is examined even though the destination is a neighbour.

The designed routing strategies are basic strategies that can be considered archetypes from which more complex and sophisticated strategies can be derived. It is chosen to design simple strategies, because it will make it easier to determine the effect and difference between the strategies.

It is expected that the application of other route strategies than the shortest path used by DSR, will lead to a higher latency. This increase in latency is the expected prize of an increase in throughput.

Routeselection strategy 1

The first route selection strategy will be to return the average of all nodes on the route.

$$T_x(Route) = \frac{\sum_{1}^{n} T_x(y_n)}{n}$$

Equation 4-3

Where $y_1, y_2 \ldots y_n \in$ Route

Using the average presents the issue, illustrated in Table 4-1, that routes containing nodes with very low trust values might still be rated high.

Route	Trust ratings for nodes on the route				Rating
1	0.7	0.7	0.5	0.5	0.6
2	1	1	-1	1	0.75

Table 4-1: Evaluation of routes using the average of nodes trust value to evaluate the route

Table 4-1 illustrates an example where two routes are evaluated by route selection strategy 1. Route consists of four nodes with values between 0.7 and 0.5. With an initial trust value below 0.5 this would mean that the node evaluating the route have had positive experiences with all nodes. Route 2 consists of four nodes where three have maximum trust value and one has minimum trust value. That the node has minimum value indicates that it is a malicious node. However, as the Rating column shows route 2 is rated higher than route 1. The example illustrates an extreme case and it is difficult to predict how often such a situation occurs. By using Equation 4-2 with a negative experience with value –1, the trust values change to the values shown in Table 4-2, which results in route 1 having a higher rating than route 2. This illustrates that the there is a fast response to experiences.

Route	Trust ratings for nodes on the route				Rating
1	0.7	0.7	0.5	0.5	0.6
2	0.8	0.8	-1	0.8	0.35

Table 4-2: The values from table Table 4-1 after having had a negative experience with value –1 and route 2

Routeselection strategy 2

The second route selection strategy is an extension of the first. In order to favor shorter routes the average of the trust values, is divided by the number of nodes.

$$T_x(Route) = \frac{\sum_{1}^{n} T_x(y_n)}{n^2}$$

Equation 4-4

Where $y_1, y_2 \ldots y_n \in$ Route

This strategy will favor shorter routes from longer, as illustrated in Table 4-3.

Route	Trust ratings for nodes on the route				Rating
1	0.7	0.7	0.7	#	0.23
2	0.7	0.7	0.7	0.7	0.175

Table 4-3: Route selection strategy 2 favor shorter routes

The strategy does still present the issues that strategy 1 has. In a domain where short routes are more frequent, this strategy might result in a lower latency than strategy 1.

Routeselection strategy 3

Instead of using the trust value of the nodes, routing strategy 3 evaluates the nodes based on the average value of the past experiences. The size of the experience window, which is the number of experiences that are remembered, will be set to 5. This has the advantage that nodes with a high trust value that suddenly starts to drop packages will be identified faster than by using the trust value. This is illustrated in Table 4-4.

Experience nr	Experience value	Trustvalue calculated based on initial trust of 0.5 using Equation 4-2	Average of the experiences
1	1	0.55	1
2	1	0.60	1
3	1	0.64	1
4	-1	0.47	0.5
5	-1	0.32	0.2

Table 4-4: Route selection strategy 3 will detect a good node that starts to drop packets fast

The table illustrates that the use of the average of experiences will identify a good node that starts to drop packets faster than the use of trust value. After three positive and to negative the average of the experiences is 0.2 where the trust value of the node, calculated using Equation 4-2, is 0.32. However, routing strategy 3 requires more computations compared to strategy 1 and 2 because it uses the experiences and not only the trust values.

Routeselection strategy 4

Route selection 4 is applied to strategy 3 but can be applied to all other strategies. Instead of returning the average a threshold is used. If a node has a value below some threshold value the route is rated with the lowest possible value. This will have the advantage that routes with low trust values are not used. It will however also have the disadvantage that the highest rated route might is discarded in the case where all routes have a rating below the threshold.

Routeselection strategy 5

This strategy returns the lowest trust value of the nodes on the route and thereby the route is evaluated based on the least trustworthy node.

$$T_x(Route) = \min\left(T_x\left(y_1, y_2 ....y_n\right)\right) \qquad \text{Equation 4-5}$$

Where $y_1, y_2 \ldots y_n \in$ Route

This has the advantage that the route with the highest of the lowest trust values is used. However, it has the disadvantage illustrated in Table 4-5.

Route	Trust ratings for nodes on the route				Rating
1	-0.3	-0.3	-0.3	-0.3	-0.3
2	1.0	0.9	0.9	-0.35	-0.35

Table 4-5: Example of routing strategy 5

As seen strategy 5 will select route 1 over route 2 even though all nodes on route 1 has low trust values.

4.1.4 Trust management

The trust manager module stores trust information about all known nodes during run time, and offers methods to query for information about stored trust values. It also functions as the main interface between the existing implementation of the DSR protocol and the trust updater and trust formatter module. In a real life scenario it is likely that nodes will move about in the same environment for some time. This means that the same nodes can be encountered on a regular basis. For this reason the trust management module implements IO methods for storing trust values in a persistent way so they can be loaded again.

4.1.5 Acknowledgement monitoring

As described in section 3.2 it is necessary to use an acknowledgement mechanism to base trust updating on. In general acknowledgements leads to a packet overhead that of course should be minimized. One way to minimize the packet overhead is to limit the amount of acknowledgements send, by using a sliding window mechanism [Sharp]. To keep it simple it is decided not to use a sliding window mechanism but instead require acknowledgements for all data packets and not for protocol packets.

The purpose of the acknowledgement mechanism is to use received acknowledgements or lack of acknowledgements to adjust trust values and not, as known for many other protocols, to base decisions on retransmission on. Since the trust values are used to base routing decisions on, and because a node can be part of many routes it is important that a missing acknowledgement is detected fast.
A short time frame might cause routes that where simply slow, but forwarded data packets and acknowledgements, to be rated low. On the other hand a large timeframe might result in a bad route being used several times before it has its trust values decreased.

An acknowledgement id is stored when the packet is send, and if an acknowledgement is received within the time frame nodes on the stored route have their trust values updated. Trust relationship should be formed with any unknown nodes on the reply route and known nodes on the reply route should also have their trust values updated. If a requested acknowledgement is not received within some time frame, the packet is considered dropped. In this case the nodes trust values should be adjusted in a negative way.
The time it takes for a packet to reach its destination will depend on the length of the routes.

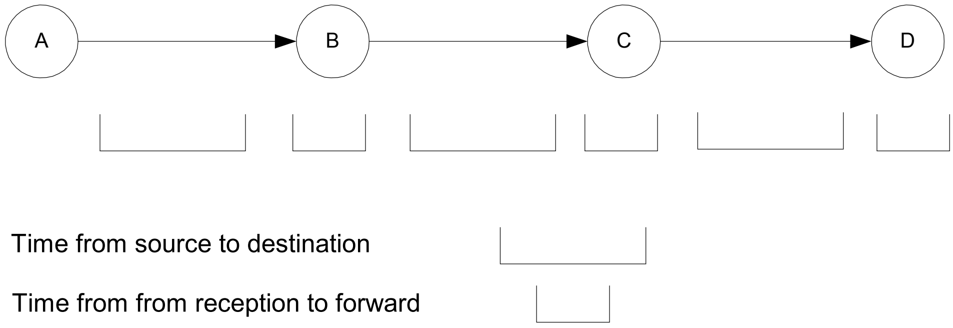

A
B
C
D
Time from source to destination
Time from from reception to forward

Figure 4-3: Estimation of total timeout value

As Figure 4-3 illustrates the number of links a packet will pass is one less than the number of nodes on the route. Since the return route of the acknowledgement will depend on the destinations route selection it is unknown how many nodes it will include and therefore the double of the length of the outgoing route is used as the ebst estimate. The following formula for estimating the total time that a node will wait for an acknowledgement will be used.

$$TO = (2L - 2) * tc \qquad \text{Equation 4-6}$$

Where TO = Total time out value, *tc* = Time out constant

The time the packet will spend on the actual physical wire is considered small compared to the time it will take for a node to process it and therefore these two times has been combined to one timeout constant. Since the node will not be in a state where it is waiting to receive the acknowledgement before it can continue, it is expected that a relatively high timeout value can be accepted. Simulations used to estimate an appropriate value for the timeout constant is described in section 6.5.2.

Calculation of the time out value means extra processing. If the system were to be used on a device where the available CPU where limited, it might be considered to cache timeout values for the most common route lengths or simply use a fixed value. This is however not done in the implementation.

4.1.6 Combining the trust modules

Since most modules need access to the trust values the trust manager module offers methods that can be accessed by the other modules for this purpose.

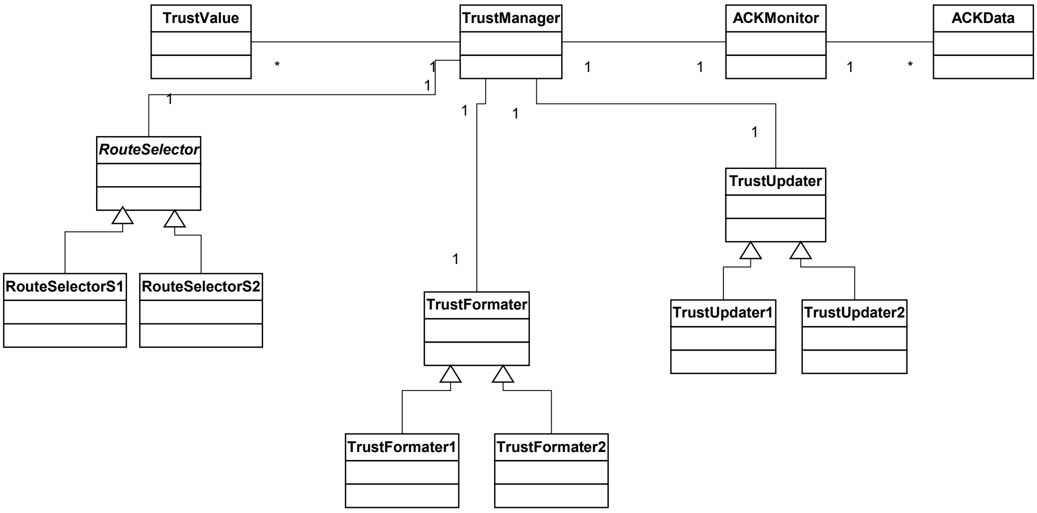


Figure 4-4: Class diagram of the trust related modules

It should be possible to change the different strategies by changing a minimum of the code. The Strategy design pattern [Gamma] is suitable for such designs since its application makes it possible to change modules/strategies during run time. Therefore it is used for the TrustUpdater module, the TrustFormation module and for the RouteSelector module. Decisions of which strategy to use, can be made at run time by an application that use the protocol, based on analysis of parameters such as the number of timed out acknowledgement requests (which indicates the number of dropped packets), often used nodes trust values, etc.

Since no such application is designed for this project the strategies are selected before a simulation is started.

4.1.7 Existing DSR Implementation in NS-2

As mentioned earlier the Ns-2 simulator is used to simulate the protocol. This simulator already offers an implementation of the DSR protocol. In order to ease the implementation of the design, an analysis of the existing DSR implementation is presented in this section. Since the implementation of the DSR protocol has a considerable size, the analysis presented here only covers the most basic functionality and classes. Figure 4-5 is a class diagram of the Ns-2 implementation of DSR that shows the most relevant classes and their relations.

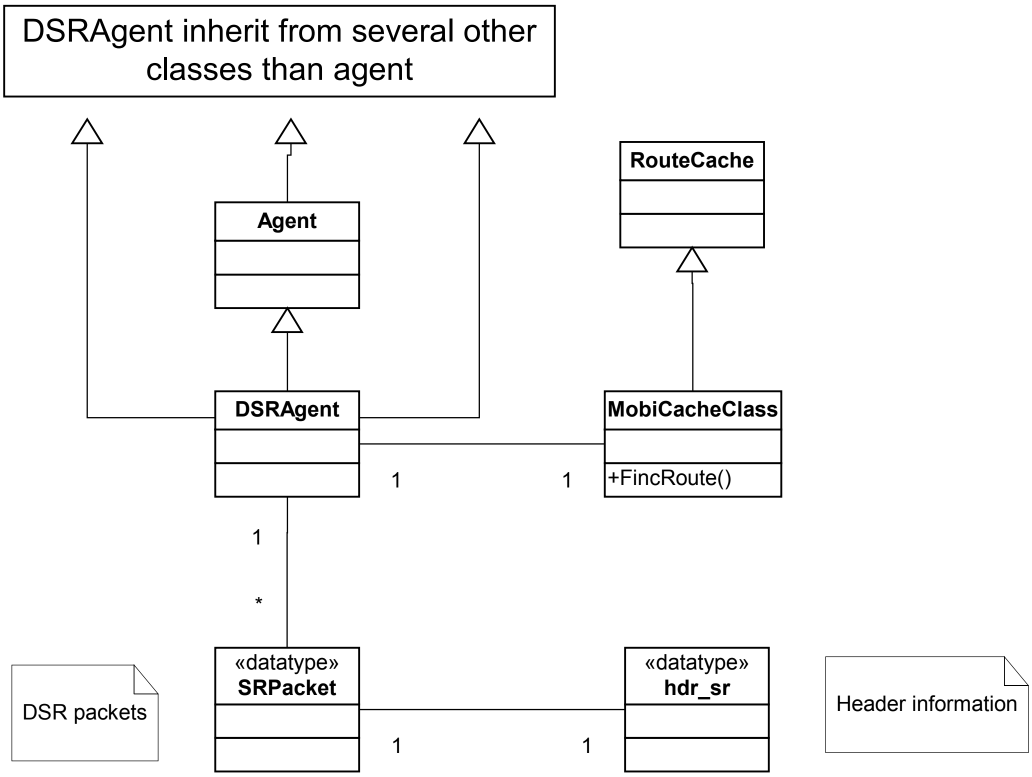


Figure 4-5:Class diagram of the most relevant classes used in the Ns-2 DSR implementation

Figure 4-6 illustrates the process that takes place when a packet is received. First it is determined if it is an incoming or outgoing packet. If it is an outgoing packet it needs a source route to the destination. To obtain a source route the route cache is queried. If a route is returned the packet is placed in a queue where it waits to be sent of by the underlying layer (the interface queue). If no route is available in the route cache a ROUTE REQUEST is broadcasted. If the packet has a source route it means that it has either reached its destination or it needs to be forwarded. If the packet has reached its destination its type is determined. There are several possible types of packages but only the ROUTE REPLY is included in the diagram since it is the most relevant. When a ROUTE REPLY is received the route is added to the cache.

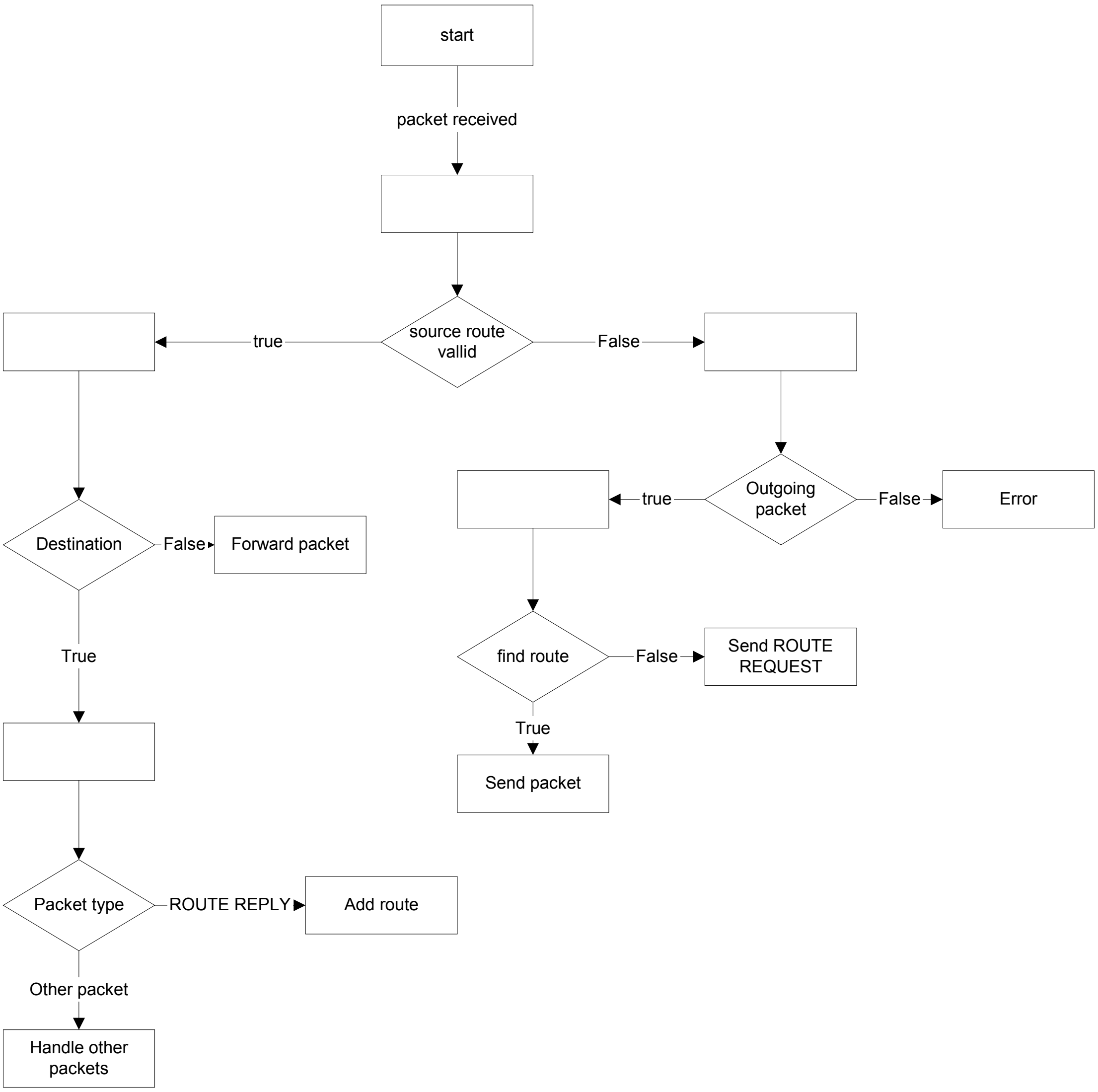


Figure 4-6: Illustration of the flow that occurs when a packet is received

The process from Figure 4-6 involves corporation between several classes. The main branching is performed in the `DSRAgent` class where the `recv()` method is called upon receipt of packages. The packet header is implemented in the `hds_sr` class and methods from this class are used to examine if the packet has a route and which type of packet it is. When the type of the packet is determined a method called `handleXXX`, where `XXX` indicates the type of packet, is called to handle further processing of the packet. The `Mobicache` class is used to store routes and offers methods to add and get routes.

4.1.8 Merging the trust modules with the existing DSR code

The trust extensions interface with DSR in the following situations:

- In order to implement the acknowledgement mechanisms extra fields to indicate whether the packet is an acknowledgement or an acknowledgement receipt has to be included in the `hdr_sr` class.

- When a packet is received and it has been determined that the packet has reached its final destination the `handlePacketReceipt()` method in

`DSRAgent` is called. Therefore processing of acknowledgement request is handled here. Further a method for constructing and sending the acknowledgment packet has to be implemented.

- Every time a route is added to the route cache by a call of the `addRoute()` method the `TrustManager` has to be called to examine the route for previously un-encountered nodes and form initial trust relationships according to the strategy implemented in the `TrustFormater` class.

- When the route cache is queried for routes the `findRoute()` method in the `MobiCache` class is called. This method is altered so the route selection strategy defined in the `RouteSelector` class is used. The `findRoute()` methods is also used to find routes that are returned as answers to ROUTE REPLY's. Introducing trust based route selection to this DSR implementation therefore has the side effect that a node will return the route it evaluates as the most trust worthy.

- When a packet is being forwarded the `handleForward()` method in the `DSRAgent` class is called. In this method malicious behavior is implemented to force packet drops.

Figure 4-7 shows a class diagram of the combined classes.

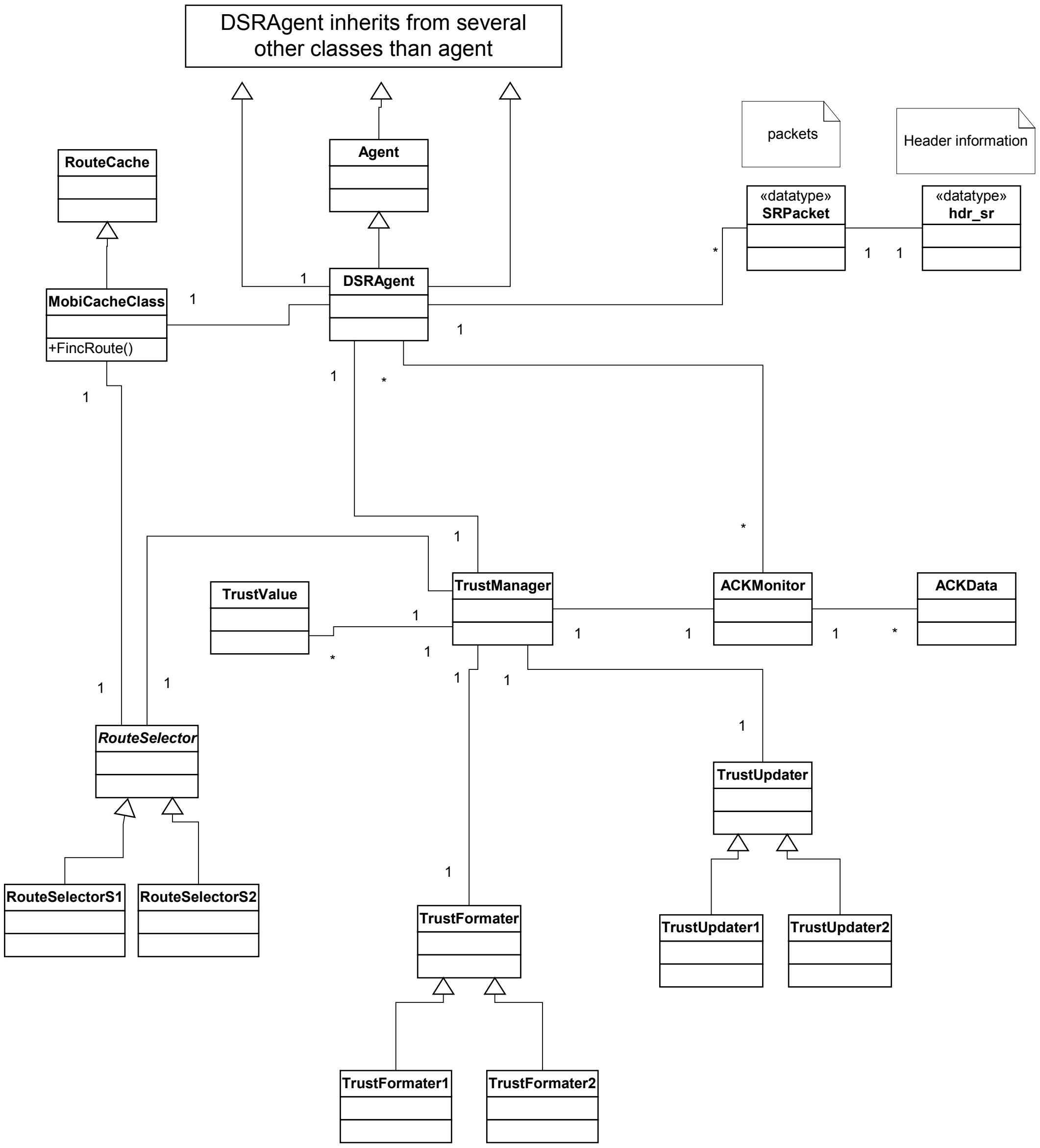


Figure 4-7: Class diagram of the combined NS-2 implementation of DSR and the trust classes.

There are three major sequences that involves interaction between several classes:

- Adding a route and initializing trust values
- Processing a received acknowledgement
- Selecting a route

Figure 4-8 illustrates the different methods that are called during the process of adding a route to the cache. When the `DSRAgent` discovers a route by either snooping it or receiving a route reply it calls the `addRoute()` methods in the route cache to add the route. After this it calls the `initTrustValues()` method in the trust manager to initialize trust values for nodes that are encountered for the first time. The `Trust Manager` propagates the call to the `TrustFormater`. The `TrustFormater` successively query the `TrustManager` for trust values for known nodes. When the minimum value is found, the `TrustFormater` successively calls the `createTrustValue()` method in `TrustManager` to create new `TrustValue` objects.

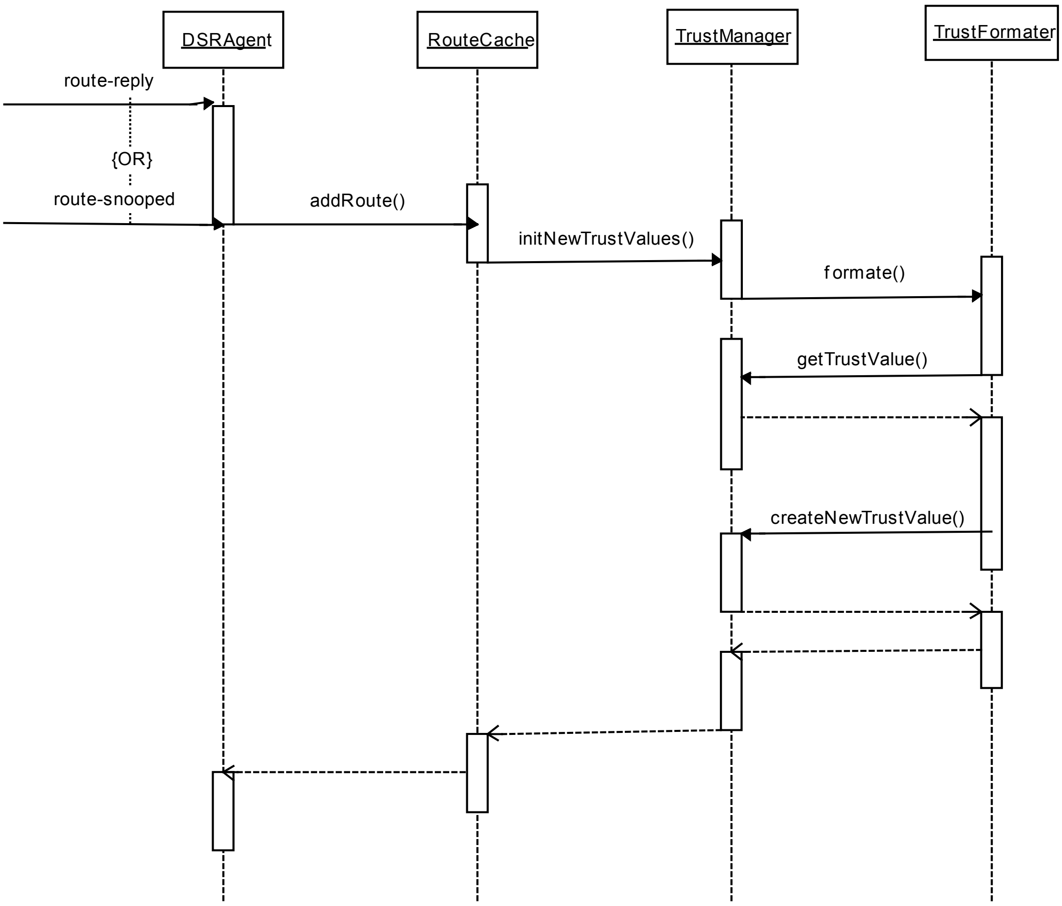


Figure 4-8: Sequence diagram describing the process for adding a new route to the route cache

Figure 4-9 illustrates the methods calls that are made when an acknowledgement is received. The `DSRAgent` examine the source header to determine if the packet is an ack. If it is, it calls the `handleACKReceived()` method in the `ACKMonitor`. If the acknowledgement request is registered it means that it has not been removed because it has timed out and therefore the `updateTrust()` method in `TrustManager` class is called. This method propagates the call to the `TrustUpdater` class by successively calling the `update()`method that updates the trust.

]

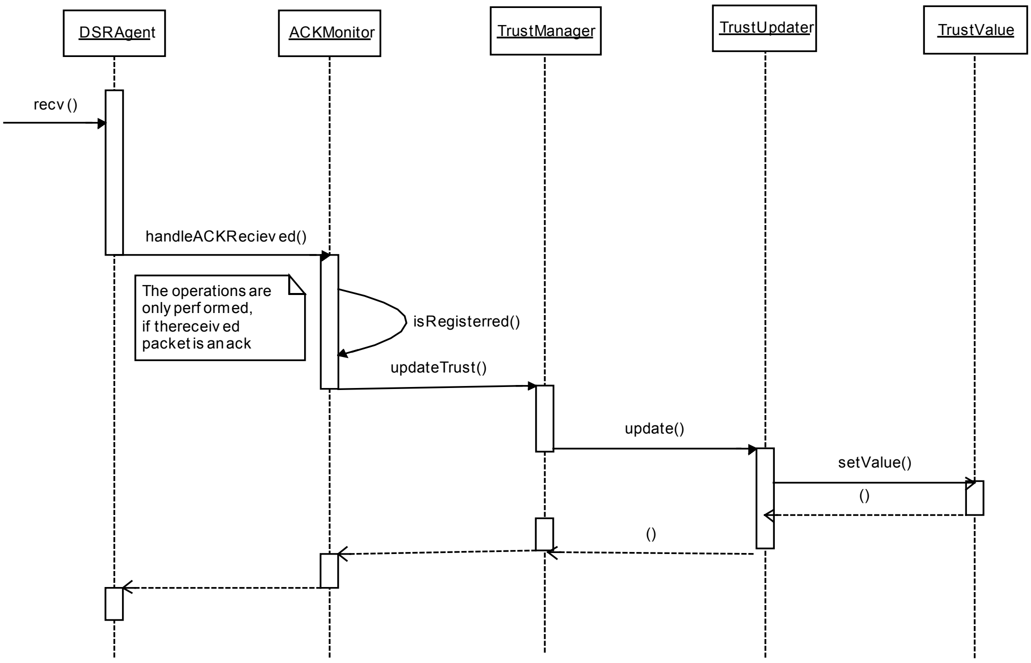


Figure 4-9: Sequence diagram describing method calls involved when receiving an acknowledgement

Figure 4-10 illustrates the method calls that are involved when a packet is being send. `DSRAgent` calls the `findRoute()` method in the route cache. To evaluate the routes the `RouteCache` calls the `evaluatePath()` method for each route to the destination. The `RouteSelector` then calls the `TrustManager` to get the `TrustValue` for the nodes on the route.

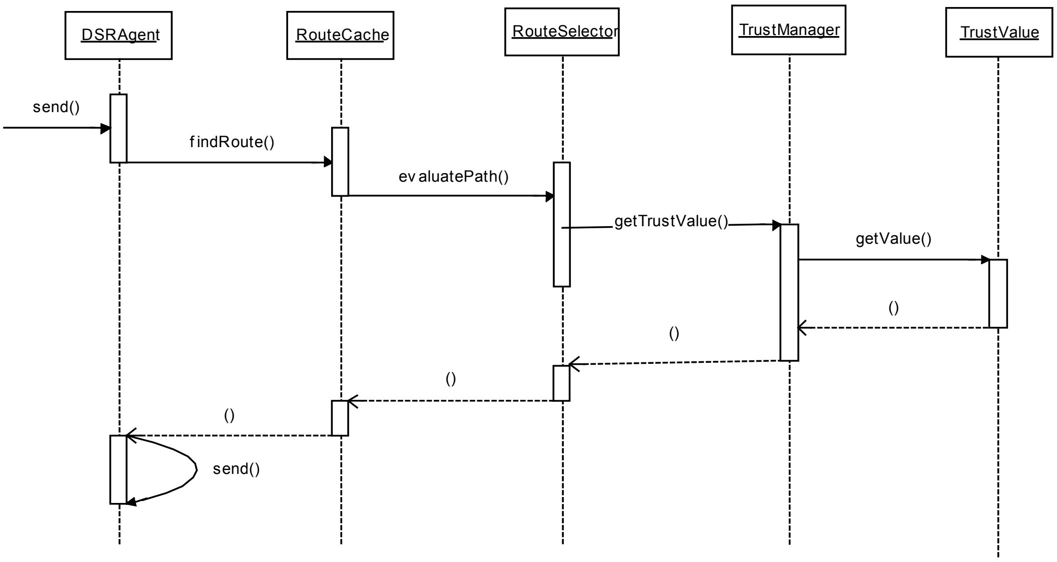


Figure 4-10: Sequence diagram describing route selection

4.2 Summary

This chapter has covered design of modules needed to incorporate trust and trust based routing in the DSR protocol, which has lead to the fulfillment of the main objective M-O 2. The following modules has been designed:

- The `TrustFormater`, that handles assignment of trust to nodes when they are encountered for the first time.
- The `TrustUpdater` that increases or decreases nodes trust values based on different strategies.
- The `RouteSelector` that will select routes using a trust based evaluation.
- The `ACKMonitor` that will handle receipt and requests of acknowledgements.
- The `TrustManager` that stores trust values for nodes and coordinates trust related processes.

Further, the implementation of the DSR protocol that exists in Ns-2 has been analyzed and methods where the trust based modules interface with the existing implementation have been identified.

5 Implementation and tests

This chapter covers the implementation of the designed protocol. The protocol builds on a C++ implementation that is included in the Ns-2 [NS2] simulator and therefore a small introduction of the Ns-2 simulator is presented in this chapter. The code has been implemented so it corresponds well to the design described in 4. Therefore it has been chosen to include specific details in appendices.

The source code for the designed trust classes are included in the appendix P. Several modifications have been made to existing DSR classes but since full credit cannot be taken this code it is not included in appendix. However, all related DSR source code is included on the enclosed CD.

5.1 Introduction to the Ns-2 simulator

This section gives a brief introduction to the Ns-2 simulator. The Ns-2 simulator has been around since 1989 and several institutions and societies has supported and contributed to its development. Ns-2 is an event driven simulator targeted especially at network research. It offers implementations of many famous and less famous protocols, traffic sources, etc.

5.1.1 Overview of Ns-2

The Ns-2 simulator consists of two major parts:

- An Otcl (Object Oriented Tcl) interpreter
- A C++ library

Implemented protocols can be accessed and used trough OTcl scripts. The OTcl scripts are used to specify node behavior such as movement and traffic generation. It is also used to specify simulation details such as simulation time, nr of nodes in simulation, protocols used for simulation, etc. The use of OTcl has the advantage that users do not need to know anything about the actual implementation. This makes it possible to start simulations with a sparse knowledge of Ns-2.

The actual implementation of the simulator and the protocols are done in C++, which means that changes or extensions to protocol behavior have to be done in the C++ code. Protocols are implemented as agents that can be thought of as part of a device that is running the protocol. This means that agents of different types can be attached thereby running one protocol on top of another. For the simulation performed in this project, a Constant Bit Rate (CBR) traffic generator agent has been attached to a `DSRAgent`. The `DSRAgent` is connected, via a link layer through an interface queue, to a Medium Access Control (MAC) layer. The link layer is connected to an Address Resolution Protocol (ARP) that finds hardware addresses that corresponds to a packets next destination. A more detailed description of the structure of the mobile node appendix I.

The manuals and tutorials mostly focus of the OTcl part of Ns-2, which means that parts related to OTcl interaction and set up is well documented. The C++ part of Ns-2 is not similar well documented and often requires examination of the source code to get information about certain topics.

Output is written during simulations to a trace file, which can be processed after simulation. Appendix N include a small description of this output.

Version of Ns-2 used in this project

The most recent released version of Ns-2 is version 2.26. This version was first used but after having experienced severe problems with DSR simulations it was discarded. Searches in the Ns-2 mail archives indicate that wireless simulations on this version are not recommended [NS2-M1]. Instead version 2.18b has been used.

5.2 Implementation details

The classes are implemented according to the design and reflect the diagram from Figure 4-7. To construct a dynamic and scalable implementation, Standard Template Library (STL) types such as maps and lists have been used. Since the implementation reflects the described design, this section will concern some specific areas of the implementation. This provides valuable information for anyone who might continue work on the code and emphasize areas where the implementation could be improved. Furthermore the code has been commented and based on these comments HTML documentation has been created with Doxygen [Heesch]. Doxygen is tool, much similar to the Javadoc, tool that extracts information from source code and generates HTML output. The generated HTML documentation is included on the enclosed CD.

The following details are included in appendices:

- A discussion of the implementation of the acknowledgement time out mechanisms. This is included in appendix O.1.
- An explanation of the implementation of malicious behavior. This is included in appendix O.2.
- A description of the file format that is used when trust values are store between simulations. This is included in appendix O.3.

5.3 Tests

This section explains the test of the implemented system that has been performed. The test of the system can be divided in to parts:

- A structural part
- A functional part

Due to the complexity of the system functional test have not been performed for all methods and classes. Because packets are created outside the DSRAgent class and pass through the class it is extremely difficult to construct packets and force events, which is necessary to perform the structural testing. Where it has been found possible small `testIt()` methods have been implemented in classes. These methods can be called at the start of simulations and write output and expected output to the screen. This output can then be verified. Since some of the test it methods create trust values they should only be called during testing, not under actual simulations because this might influence the results.

The functional tests have been performed by writing and verifying output.

Several flags have been implemented in the `TrustConstants.h` file that can be used to control which output that is written. The functional tests have focuses on verifying the following:

- That acknowledgements are send and received
- That routes are selected and evaluated
- That trust values are updated
- That malicious nodes drops packets

The output can be generated for verification by using the VERBOSE flags in the `TrustConstants.h` file.

5.4 Summary

This chapter has presented a short introduction to the Ns-2 network simulator explaining the overall architecture of the simulator. Specific details about the implementation, such as the acknowledgement timeout method and the file format for saved trust values have been included in appendices. HTML documentation of the source code have been created using Doxygen and is included on the CD. Further some of the tests of the system that have been performed have been discussed.

6 Simulations and Results

This chapter covers the simulations that have been conducted with the implemented system to meet the post objective, P-O 1. Randomness of the scenarios that have been used for the simulations are discussed, to elucidate their expected impact on the results. Several parameters have been subject to examination and the most important is accentuated. Finally some of the achieved results are presented, analyzed and discussed. Details related to the simulation platform is included in appendix L. Instead of determining the effects of trust based routing by simulations other methods such as analysis could have been used. However, due to the complex nature of the ad hoc network, analysis verification has not been considered realistic within the time frame of this project.

6.1 The randomness of simulations

This section presents a discussion of the randomness of the simulations.
There are two main parts that has an impact of the outcome of the simulations:

- The movement scenario
- The traffic scenario

Ns-2 provides tools to create both of these. To simulate movement the Random Waypoint model is used. The Random Waypoint model is a commonly used mobility model for simulations with ad hoc networks. It functions in the way that nodes pick a random destination towards which they start moving along a straight line with some maximum speed. When the destination is reached the node waits for a specified time, the pause time, before it starts moving again. The scenario is generated randomly, but once it has been generated it is deterministic which means that the topology will always be the same, with the same scenario. It is clear that the choice of which nodes should act as malicious is tightly related to the scenario.

To simulate traffic Constant Bit Rate (CBR) traffic scenarios are used. CBR is chosen over TCP because the protocol is much simpler which makes the results easier to analyze. Furthermore, it seems to be best practice to use CBR for ad hoc simulations [Broch1] [Buchegger2]. The traffic scenarios are also generated randomly, but similar to the movement scenarios they are deterministic once created. The traffic scenarios specify which nodes should start transmitting to some destination at a specific time.

Simulating random movement and traffic requires several simulations with changing scenario and traffic files, which is a very time consuming task. Since the objective of the simulations is to compare the standard DSR and DSR with trust based route selection, using one scenario is considered satisfying, as long as one protocol does not gain advantages over the other from the scenario. However, to ensure that the results are not obtained by using an exaggerated positive scenario, simulations to determine the impact of using different scenarios are carried out.

Since the outcome of a simulation is the same when the same scenario and same number of malicious nodes is used only one simulation is conducted with the DSR protocol. As mentioned in section 4.1.4, trust values are stored between simulations, which can cause nodes to make different choices of route selection during one simulation than during the previous simulation. This means that the results can vary between simulations with trust based route selection. To account for this, five subsequent simulations are performed and the trust values are stored after each simulation. When a new simulation batch of five simulations is begun with a different number of malicious nodes, the trust values are deleted so the nodes start without any stored trust information.

6.2 Metrics

There are several different metrics that can be applied to measure protocols performance against. Studies of performance evaluations of protocols for mobile ad hoc networks indicate that the following metrics is usually used [Broch1], [Buchegger2].

Throughput, which is the ratio between the number of packets send by the application layer and the number of packets received at the application layer. This is calculated by Equation 6-1.

$$Throughput = \frac{\sum_1^n received}{\sum_1^n send}$$

Equation 6-1

Path optimality, which is the difference between the number of hops the packet took and the length of the shortest path.

Routing overhead, which expresses the total number of routing packets that are send.

The throughput is used to compare the standard DSR and DSR fortified with different trust based routing strategies, because it is a very common metric and because it expresses how well the protocol is at delivering packets for an overlying (application) layer, which in the end, is the primary task of the protocol.

Using path optimality in the common way is not considered relevant because trust based routing is based on the avoidance of malicious nodes, which is expected to lead to the use of longer routes. Therefore it is not considered a primary objective to select a short route. Instead a different metric based on how well the routing strategies was to avoid using routes with malicious nodes is used.

Because the fortified DSR protocol uses the acknowledgement strategy described in section 4.1.5, all received data packets results in an acknowledgement being send, which leads to an increased routing overhead. This overhead is not determined because it was considered of less importance. For every received acknowledgement request, non-malicious nodes return an acknowledgement. However, as the packet is received, a possible route to the acknowledgement destination is discovered, which means that the number of extra routing packets, beside the acknowledgements, might be as low as zero.

Other metrics such as storage complexity, time complexity and packet size have also been defined [Zou], but have not been used.

6.3 Processing the output

As mentioned the output that is generated is mainly written to the trace file. A trace file from a simulation can consist of more than 100.000 lines and therefore a parser to process it is necessary. The code for this parser in written in Java (because it offers easy IO and string handling) and is placed in appendix P.
The process of handling and processing the data is illustrated in Figure 6-1.

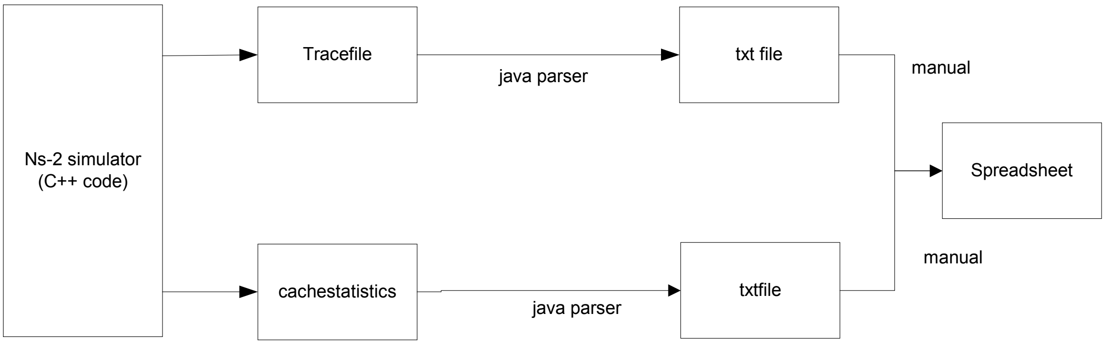


Figure 6-1: Data processing of simulation results

Besides the trace file, methods to write data from the cache have been implemented. This data that relates to the routes that are selected is written to a text file that can be preprocessed. The processed results are stored in a spreadsheet were they can be examined further. As mentioned in section 6.1 storing trust values have an impact on the results, which results in deviations. To account for these deviations, statistical methods to analyze the results are used. DSR and DSR with trust based route selection need to be compared to make a statement whether or not they are different.

An often-used method to verify that results can be used to decide that two systems are different, is by using confidence intervals [Jain]. Confidence intervals for the mean can be calculated using Equation 6-2.

$$Con = \left(\overline{x} - \frac{z_{1-\alpha/2} * s}{\sqrt{n}}, \overline{x} + \frac{z_{1-\alpha/2} * s}{\sqrt{n}} \right) \qquad \text{Equation 6-2}$$

Where: $Con = \text{Confidence interval}, \overline{x} = \text{mean}, n = \text{samplesize}$

$S = \text{Standard deviation}, z_{1-\alpha/2} = (1-\alpha/2) - quantile$

A 90% confidence interval has been used for the route selection strategies. This means that it can be stated with a 90% confidence that the mean lays within the confidence interval [Jain]. This is used when the throughput of a route selection strategy has been compared to that of standard DSR.

Due to the size of the trace files, that are the main result of a simulation, they are not included on the CD. They are stored on a drive at DTU and can be obtained by contacting the author of this thesis.

6.4 Parameters

Several parameter that can be adjusted and thereby influence the outcome of the simulations exist. This section gives a brief presentation of the most important parameters. The parameters are presented in tables that also contain information about the value of the parameter that was used during simulations and whether this value was fixed or not. The names in the Parameter column in Table 6-1 and Table 6-2 correspond to the name the parameters have in the source code.

6.4.1 Table of standard DSR parameters

Parameter	Description	Fixed	Value
Snoop_source_routes	Flag used to indicate if source routes should be snooped. The possible impact of snooping source routes is covered in section 3.1.3	Yes	False
Reply_only_to_first_ routereq	Flag used to indicate if a node should only reply to the first route reply it receives. The expected impact of this parameter is covered in section 3.1.3	Yes	False
Send_grat_replies	Flag used to indicate if a node should send out gratuitous replies to shorten routes. The expected impact of this parameter is covered in section 3.1.3	No	True/False
Reply_from_cache_on propagation	Flag used to indicate if a node should, when receiving a route request, reply with a route from its cache if possible.	No	True/False
Use_ring_search	Flag used to indicate if a node should, use ring search to discover new routes. The impact of this parameter was discussed in section 3.1.1.	Yes	False

Table 6-1: DSR parameters

Furthermore, several parameters related to route reply timeouts, buffer size, route cache size, etc. exists. These parameters have just been assigned the default value from the implementation.

6.4.2 Trust related parameters

This section covers the parameters that are related to the trust based extension. The parameters can be changed in the TrustConstants.h file. Since they have already been covered in details in chapter 4, they are only summarized here.

Parameter	Description	Fixed	Value
AVR_ACK_TIMEOUT_VAL	Average timeout value for acknowledgment	No	#
NOACKRECEIVED	Value of no acknowledgement received event	Yes	-1.0
ACKRECEIVED	Value of acknowledgement received event	Yes	1.0
DATARECEIVED	Value of data packet received event	Yes	0.7
MAXTRUSTVAL	Maximum trust value	Yes	1.0
S1_DEFAULTTRUSTVALUE	Default trust value when no nodes are known on the route during trust initialization. (S1 relates to trustformation strategy 1)	No	#

Table 6-2: Parameters related to the trust extension.

6.4.3 Other parameters

This section summarizes some parameters related to topology, traffic rates etc. These parameters are all specified in the OTcl script, which is included in appendix P.3. Most of the values have been assigned based on a decision on what was used in the documentation and the examples that have been studied [Fall], [Greis], [NSEx].

Parameter	Value
Application traffic	CBR
Radio Range	250 m
Packet Size	512 bytes
Transmission rate	4 packets/s
Pause time for nodes	60 s
Maximum speed	1 m/s
Simulation time	500 s
Number of nodes	25
Area	1000 m × 1000 m
Available bandwidth	1 Mb/s

Table 6-3: Parameters used to specify node behavior and simulation details

The speed of 1 m/s is chosen because it corresponds to slow walking. The number of nodes in the network is set to 25, which corresponds to the assumption, pointed out in section 2.1.4, that DSR operates in a network with a diameter of 5-10 nodes. For a simulation that last 500 seconds, approximately 30000 CBR packets are send. This number is considered high enough to eliminate any deviations influence on the results. A remark needs to be tied to the value of the available bandwidth. It has not been possible to find available information on the bandwidth of the mobile wireless networks that are used for the simulations in the documentation of Ns-2 [Fall]. Searches of the ns mail archives indicate that the available bandwidth is 1 Mb/s [NS2-M2]. With this bandwidth, a packet size of 512 bytes and a transmission rate of 4 packets/s, congestion of the network is not likely to occur.

6.4.4 Malicious nodes

A minimum of 0% and a maximum of 40% malicious nodes have been used. The same nodes always act as malicious, which means that if node 2 was malicious in a simulation with three malicious nodes it is also malicious in a simulation with five malicious nodes. Table 6-4 contain the nodes that were malicious during simulations. The upper row indicates which nodes were used when the total number of nodes was varied. If, for instance three malicious nodes were used, node 1,2 and 7 act as malicious. The nodes that act as malicious have been selected randomly and then kept fixed for all simulations.

Total nr of malicious			**3**		**5**		**7**			**10**
Node nr	1	2	7	10	16	19	23	13	24	5

Table 6-4: The numbers of the nodes that were malicious during simulations.

6.5 Preliminary simulations

This section presents preliminary simulations that have been conducted in order to make a first fit of certain parameters. The estimation of these parameters has not been of primary interest, which means that the fitting was simply done by trying different values and choosing the one which lead to the highest throughput.

6.5.1 Estimation of initial trust

To estimate the optimal value to assign to nodes when they are first encountered some simulations are performed. Assignment of the initial trust value occurs, as described in section 4.1.1, when a new route composed of un-encountered nodes is discovered. Table 6-5 shows the parameters that were used for the simulation.

Trust related parameters	Value
Routing Strategy	1
Average timeout value for acknowledgment	0.5
Malicious nodes	7

DSR parameters	Value
Send_gratiuos_replies	True
Reply_from_cache_on propagation	True

Table 6-5: Parameters used when estimating the optimal value of initial trust.

The results from the simulation are presented on Figure 6-2. As seen, the highest throughput just above 48% is achieved when initial trust of -0.4 is used.

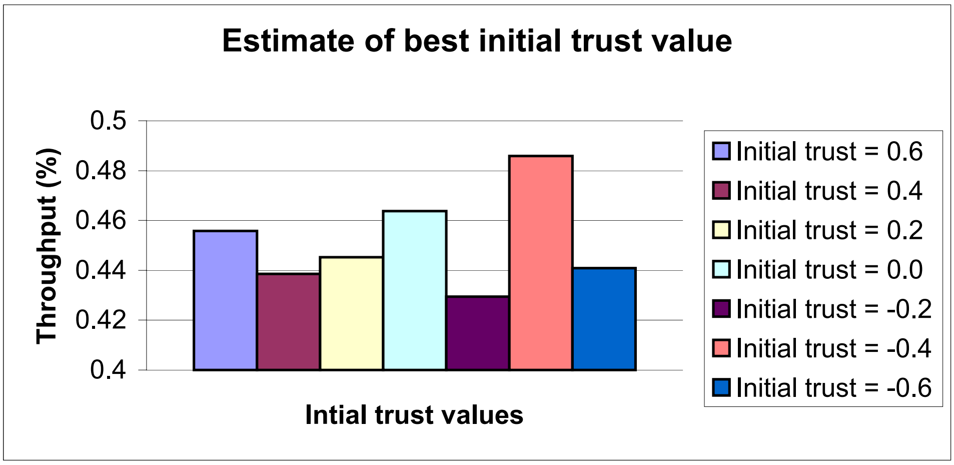


Figure 6-2: Results from running simulations with different values of initial trust.

The purpose of the simulations was merely to estimate a value of initial trust. The results indicate that -0.4 results in the highest throughput and therefore this value is used for the remaining simulations. A value of -0.4 means that the node is initially distrusting which in a scenario with 28% malicious nodes is considered sensible.

6.5.2 Estimation of acknowledgement time out

The importance of the time that the acknowledgement monitor should wait before treating an acknowledgement as not received is discussed in section 4.1.5. To estimate a suitable value for this parameter simulation were carried out using the settings from Table 6-6.

Trust related parameters	**Value**
Routing Strategy	1
Malicious nodes	7

DSR parameters	**Value**
Send_gratiuos_replies	True
Reply_from_cache_on propagation	True

Table 6-6: Parameters used when estimating acknowledgement time out value

Figure 6-3 below shows the results of the simulation.

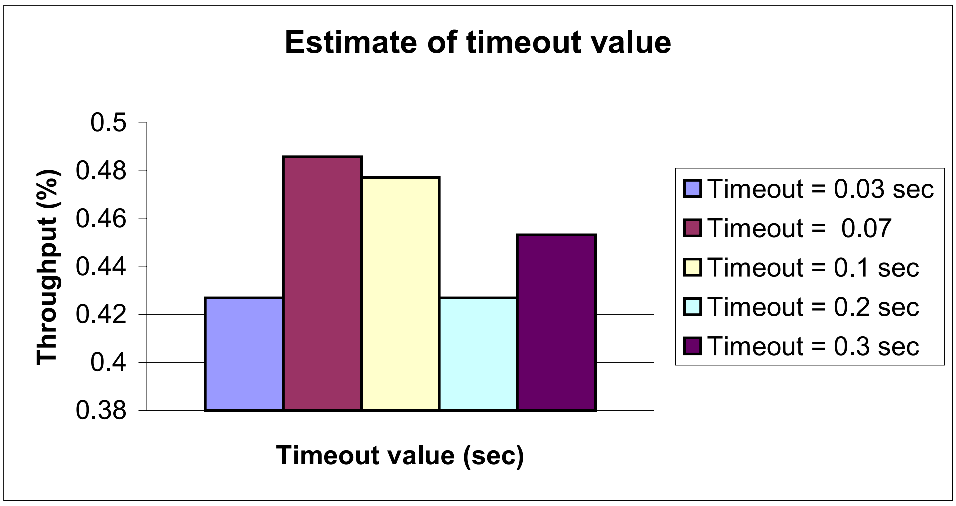


Figure 6-3: Results from estimation of acknowledgement timeout value

As the figure shows the highest throughput is attained with a timeout value of 0.07 seconds. By using Equation 4-6, this means that for a route of length 4 a node will wait 0.42 seconds before the request of an acknowledgement is forsaken. Considering that a transmit ratio of 4 packet/s is used, this means that in the case with a wait time of 0.42 second only 2 packets can be send by a route before a possible acknowledgement time out occurs and the routes trust values adjusted. For the remaining simulations an acknowledgement timeout value of 0.07 seconds has been used.

6.5.3 Impact of using different scenarios

To investigate the effect of using different scenarios, simulations have been performed with the parameters listed in Table 6-7. Five different traffic scenarios and five different movement scenarios were used for the simulations.

Trust related parameters	**Value**
Routing Strategy	1
Malicious nodes	7

DSR parameters	**Value**
Send_gratiuos_replies	False
Reply_from_cache_on propagation	False

Table 6-7: Parameters used during simulations with different scenarios.

Figure 6-4 displays the results of the simulations.

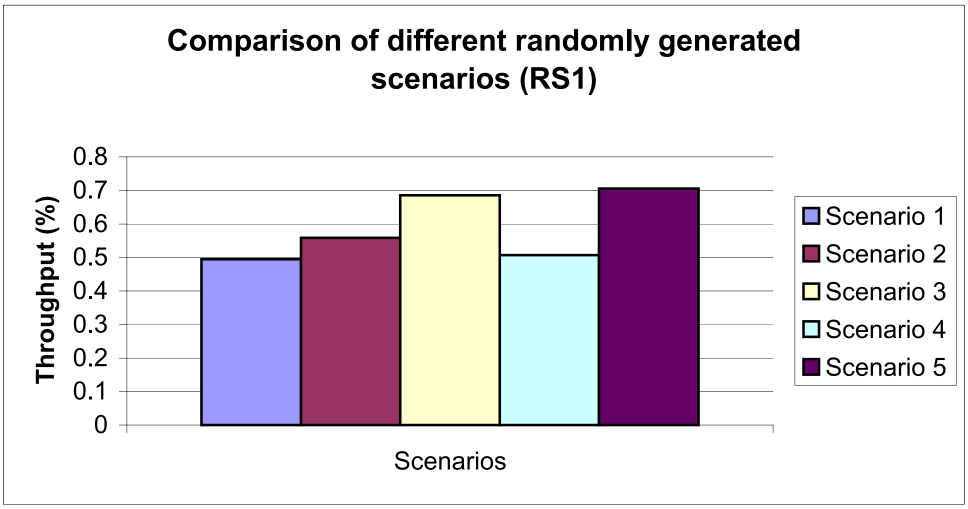


Figure 6-4: Results from different scenarios.

As seen there is a significant difference between the results. The lowest throughput of approximately 50 % is experienced with scenario 1, and the highest, of approximately 70% with scenario 5, which leads to a difference of more than 20 % between the best an the worst result. To examine the cause of this significant disparity the nodes physical position during the simulation is examined. The nodes positions at the start of the simulation are presented on Figure 6-5 and Figure 6-6. The figures are only sketches, which means that the scaling does not hold. The area is 1000 m^2 and nodes move with a maximum speed of 1 m/s with a pause time of 60 seconds. The simulations were run for 500 seconds. This means that a node as the maximum can have moved 500 m during the simulation. Malicious nodes are marked with red on the figure.

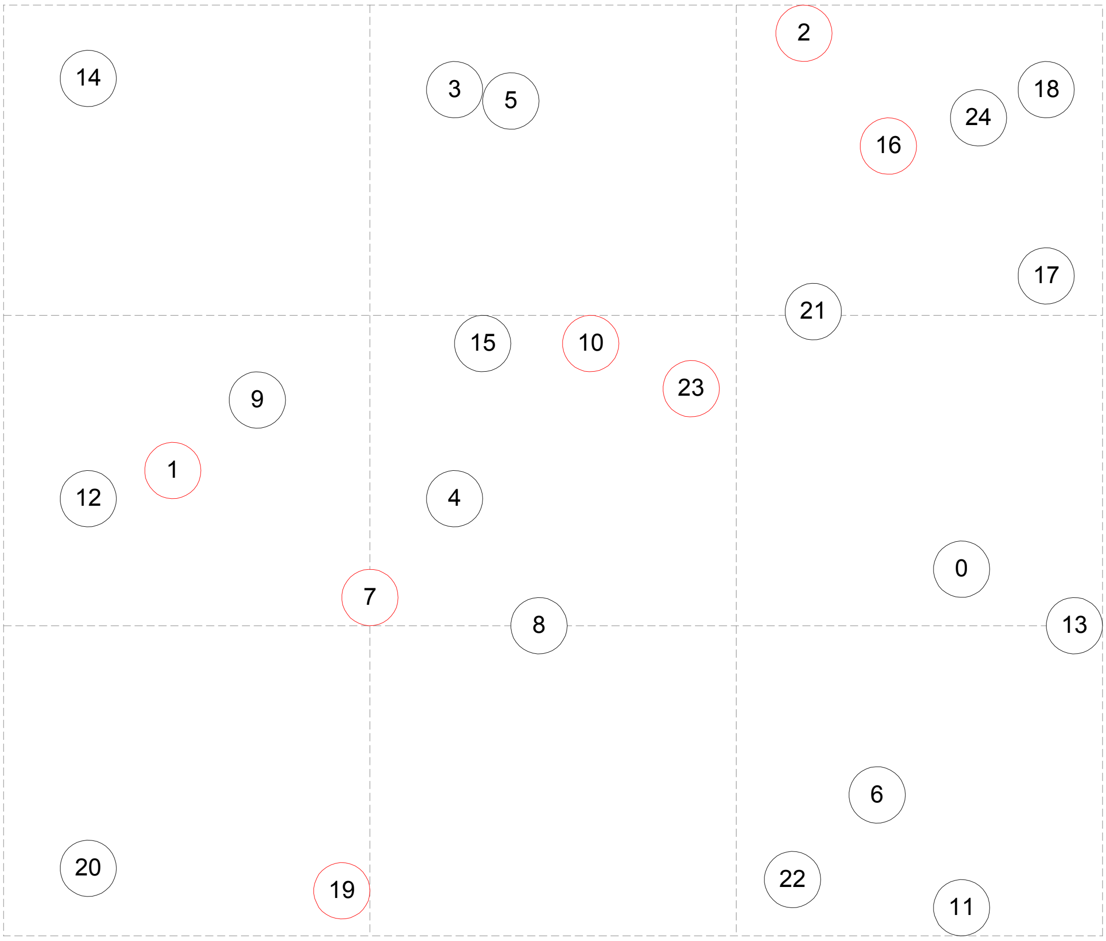


Figure 6-5: Nodes position at start of simulation with scenario 1.

As seen on Figure 6-5 the mid section contains three malicious nodes. It is likely that any packets that are being send across the area in some way is likely to pass over the mid section. This means that the malicious nodes will most likely be on many of the routes, which can explain the low throughput. In Figure 6-6 it can be observed that no malicious nodes are present in the mid sections in scenario 5 and furthermore, all malicious nodes except one, are located in the rightmost section. This means that packets send from the leftmost section to the middle section or from the middle section to the left section is likely to be transported by routes with no malicious nodes. Further more, packets send from the right section will also be likely to reach their destination since the risk of encountering a malicious node in the mid or left section is low. It is clear that the number of nodes that are only one hop away from a malicious node is larger in scenario 1 than in scenario 5 since the malicious nodes in scenario 5 is placed close to the border of the area. Being one hop away from a malicious node increases the risk of being part of a route with that node and also increases the risk of routing over that node. These considerations can also be used to explain the difference in the throughput for the two scenarios.

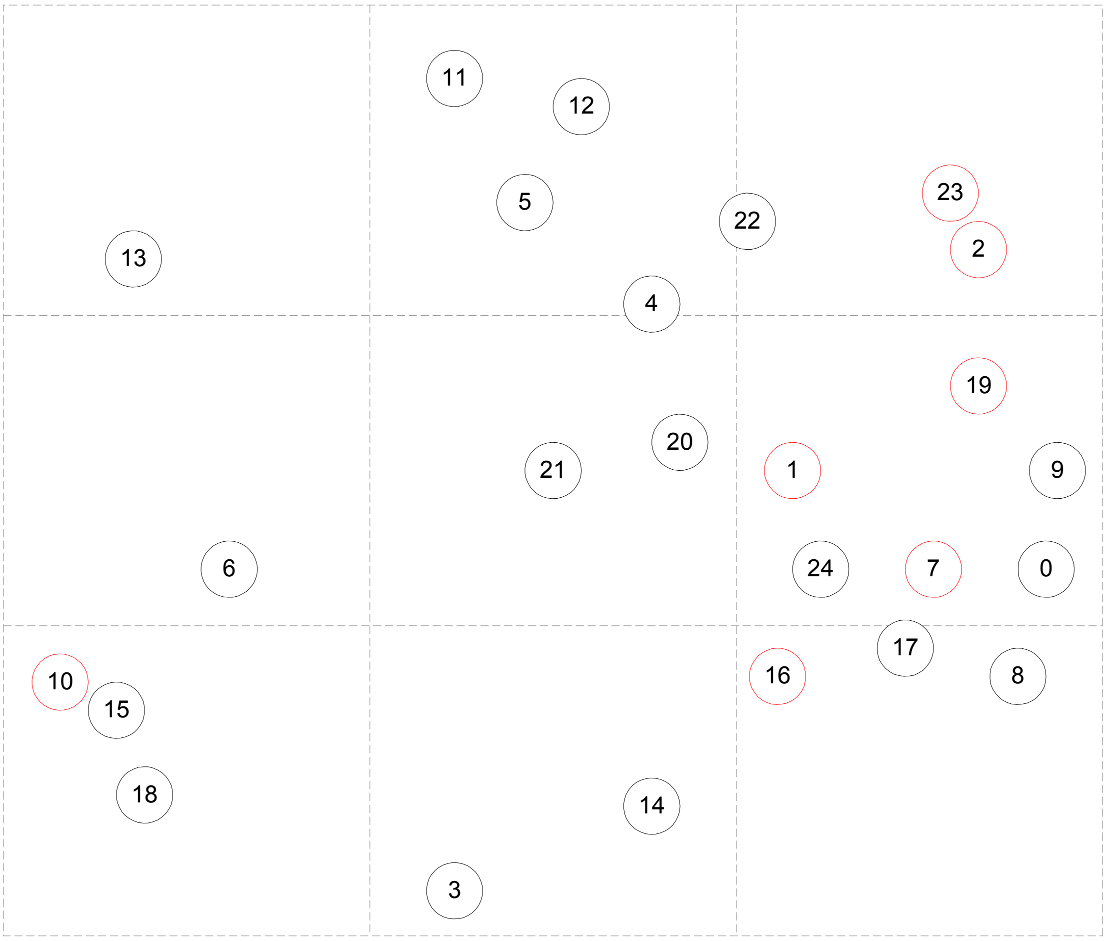


Figure 6-6: Nodes position at start of simulation with scenario 5

Whether or not it is realistic with a scenario where the nodes are positioned as in scenario 5 is difficult to make a statement about, but to take a pessimistic view of the results and consider the worst-case scenario, scenario 1 is used for the simulations.

Simulations with standard DSR are performed with the same scenarios and the results presented in appendix K. These results show that an increase in throughput of approximately 20 % could have been achieved by using scenario 5.

6.6 Comparison of Route selection strategies

This section presents and discuss results and simulations related to the comparison of route selection strategies. The throughput is compared to determine the influence of the trust based routing compared to standard DSR. The first results were obtained by simulating with the parameters from Table 6-8.

Trust related parameters	Value
Routing Strategy	1
Malicious nodes	0-40 %

DSR parameters	Value
Send_gratiuos_replies	True
Reply_from_cache_on propagation	True

Table 6-8: Parameters used for first comparison of route selection strategy 1 and standard DSR

The values used for the DSR protocol simulations are presented in Table 6-9. These values have been used for all simulations, and were chosen simply because they are default.

DSR parameters	Value
Reply_only_to_first_ routerequest	False
Send_gratiuos_replies	True
Reply_from_cache_on propagation	True
Use_ring_search	True

Table 6-9: Parameters used for the DSR protocol

The resulting throughput derived from the simulations is presented on Figure 6-7. The two curves following the RS1 curve are lower and upper confidence intervals.

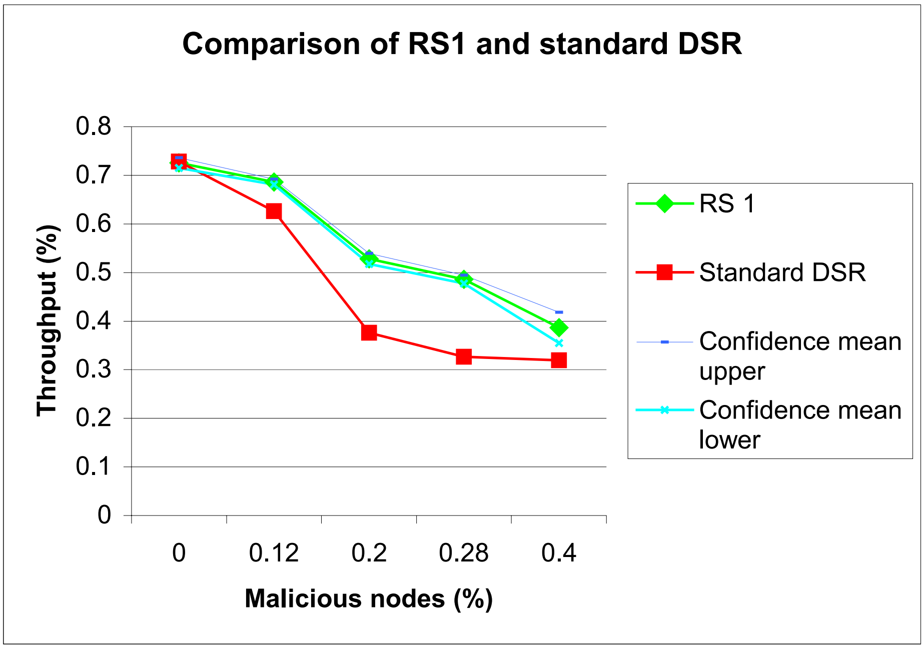


Figure 6-7: Comparison of route selection strategy 1 and standard DSR

The first noticeable thing is that both standard DSR and route selection strategy 1 only manage to deliver around 72 % of the packets even though no malicious nodes are present in the scenario. This result is somewhat surprising compared to some of the results mentioned in section 2.1.6 where DSR is said to deliver as much as 95 % of the packets [Broch1]. However, Broch et al actually identifies a serious layer integration problem with the ARP mechanism that, as they state, "*would affect any on demand protocol running on top of an ARP implementation similar to that of BSD Unix*" [Broch1 pp 11]. The problem occurs when a series of packets with a next-hop destination whose link-layer address is unknown are passed to the ARP. This result in all packets, but the last added, being dropped by the ARP. Johnson et al implemented a solution to this problem, but it is considered out of scope of this assignment to implement a similar solution. The most recent version of the Ns documentation states that the ARP in Ns-2 is implemented in BSD Unix style [Fall]. This is most likely the explanation of the low throughput experienced for standard conditions. Due to these circumstances comparison of throughput might not reveal the full effect of trust based route selection, because it cannot be determined if the packets dropped by the ARP, were to be send by route with or without malicious nodes.

Further more, these observations indicate that achieving a throughput much above 70% is most likely unrealistic.

As expected the throughput decreases as the number of malicious nodes in the network increases. As seen the throughput for standard DSR and route selection strategy 1 approach each other when the number of malicious nodes moves towards 40%. The interval beyond 40 % malicious nodes is not examined because the results clearly indicate that the effect of trust based routing decreases beyond this point. With 28 % malicious nodes the throughput is approximately 50 % for route selection strategy 1, which is a 15% improvement compared to standard DSR.

Figure 6-8 shows the results of simulations with the five different routing strategies that were covered in section 4.1.3. Route selection strategy 4 used a threshold value of -0.25. It was chosen to apply -0.25 as threshold value because with an initial value of -0.4 a value of -0.25 would indicate that the node had been part of a positive experience or part of a route with a node that had resulted in a positive experience.

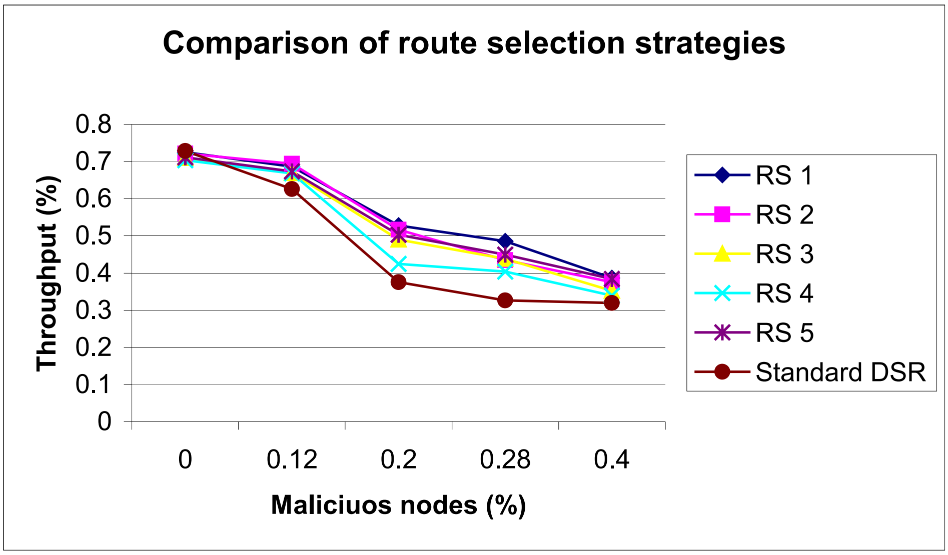


Figure 6-8: Comparison of different routing strategies.

The figure illustrates that all routing strategies offers a higher throughput than standard DSR. Especially in the interval 20% - 28% malicious nodes is the difference markedly. The maximum difference is approximately 15 % between standard DSR and route selection strategy 1 at a level of 28% of malicious nodes.
Routing strategy 4 performs worse than the other strategies when the percentage of malicious nodes increases. When the number of malicious nodes increases there is a higher probability that good nodes will participate in routes with malicious and thereby be part of a route that drops a packet. This means that good nodes can also (momentarily) be assigned low trust values. If no routes have nodes without trust values below the threshold value, the last available route will be selected due to the way the route selection is implemented, which can result in a route with minimum trust value being used. As seen the difference between the other routing strategies are not that significant, but routing strategy 1, 2 and 5, with 1 as the best, seems to perform slightly better than routing strategy 3. Since routing strategy 3 and 4 are related it is expected that their performance to some extent are similar.

The results are however, so close that the confidence intervals overlap, which means that it cannot be concluded with a 90% certainty that the mean lies between the interval. This means that it is with some uncertainty that it, based on these results, can be concluded that one strategy is better than the other.

There are many other factors besides malicious drops that can cause packets to fail to reach their destination. One factor is the ARP protocol, which was mentioned earlier. It has also been observed that are large number of packets are dropped internally in the node from the Sendbuffer. Data and routing packets are stored in the Sendbuffer if no route is found to their destination. They are dropped from the buffer after a certain amount of time or if the buffer is full. Appendix M shows a graph of the drops from the Sendbuffer, which occurred during simulations with route selection strategy 1. The results are similar with other routing strategies and for standard DSR. It is expected that it becomes more difficult to discover routes as the number of malicious nodes increase, because malicious nodes do not forward any type of packets. This can explain the increase in sendbuffer drops. However, no good explanation can be given to why so many packets are dropped when no malicious nodes are present. The packets drops caused by other factors makes it difficult to analyze the results, because it cannot be decided whether a packet where to be forwarded over a good route. Therefore no further results related to comparison of throughput are presented.

6.7 Evolution of trust values

In this section it is investigated how well the nodes identify malicious nodes. This is done by saving the content of each node's TrustManager after each simulation. For these simulations the parameters from Table 6-10 were used.

Trust related parameters	**Value**
Routing Strategy	1
Malicious nodes	7
Initial trust	0.5

DSR parameters	**Value**
Send_gratiuos_replies	True
Reply_from_cache_on propagation	True

Table 6-10: Parameters used for simulations for analyzing trust values evolution.

Figure 6-9 shows how the nodes trust values evolve. The trust values are calculated as an average of all the nodes trust values in each node. The figure clearly illustrates that the seven malicious nodes: 1, 2, 7, 10,16, 19 and 23 all have low trust values and that the trust value has decreased for each simulation. Node 5, 15 and 24 does also have relatively low trust values even though they are not malicious. This can, to some extent, be explained by studying Figure 6-5 that shows the nodes position. It can be observed that node 15 is positioned in the mid section very close to the malicious nodes 10 and 23. Node 24 is placed in the upper right section very close to the malicious nodes 2 and 16. Being close to malicious nodes increases the risk that a node is on a route with this malicious node, which means that the node can be part of a negative experience for other nodes. This is believed to be the case for nodes 15 and 24. Node 5 is not, as node 15 and 24, particularly close to any malicious nodes, which make it a bit more difficult to explain why it has a low trust rating. The explanation might be sought in the fact that all nodes do not send packets to all nodes. This means that some nodes do not have experiences with other nodes and that nodes might not be part of good routes. If traffic for node 3 comes from the rightmost section node 5 might be part of the route. If malicious nodes drops traffic for node 3 and node 5 is part of the route it will have its trust value decreased. This might be what has caused node 5's low trust value. However no empirical data exist to support this hypothesis.

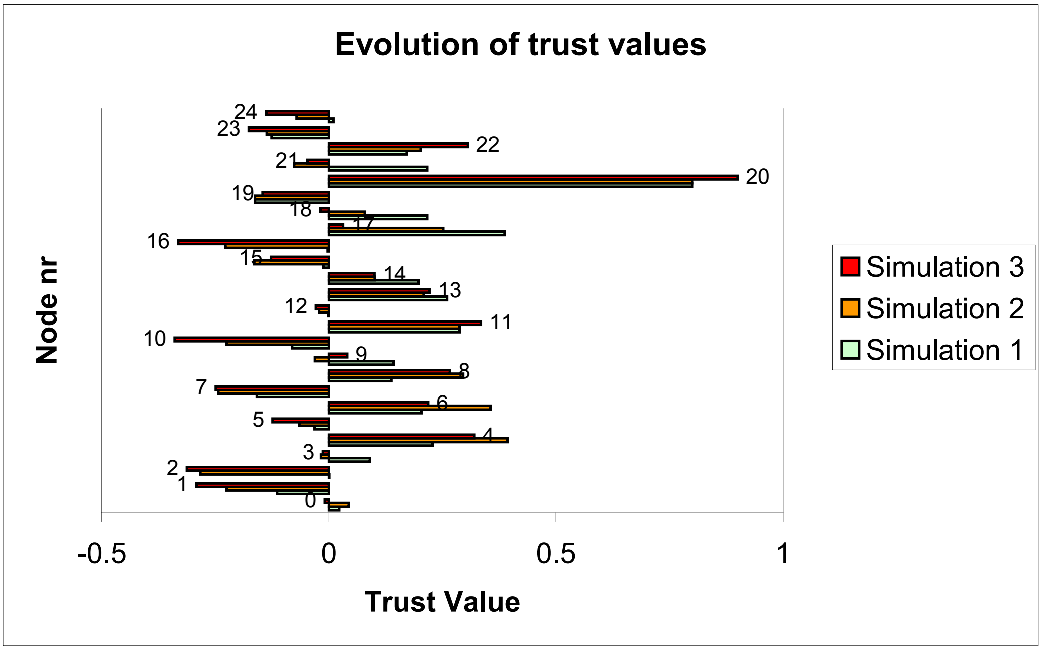


Figure 6-9: Evolution of trust values

The nodes ability to identify malicious nodes corresponds well to the results that Keane achieved with his implementation of trust based routing [Keane].

6.8 Malicious packet drops

To determine the amount of packets that are dropped by malicious nodes further simulations have been carried out. For these simulations the parameters in Table 6-11 have been used.

Trust related parameters	Value
Routing Strategy	1
Malicious nodes	7

DSR parameters	Value
Send_gratiuos_replies	False
Reply_from_cache_on propagation	False

Table 6-11: Parameters used for simulations to measure the number of malicious drops

The results from the simulations are presented in Figure 6-10.

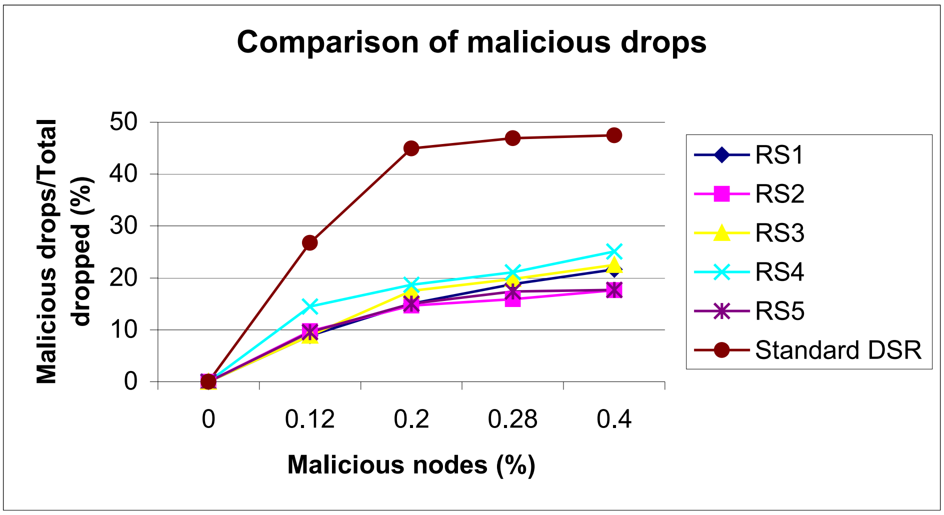


Figure 6-10: Comparison of drops caused by malicious nodes

The graphs express the number of packets dropped by malicious nodes divided by the total number of packets. Packets dropped by malicious nodes are both data packets and routing packets where the total number of dropped packets are only routing packets. This is done because it at the time of simulation was not possible to differentiate the two. Later measurements have showed that approximately 80% of the malicious drops are data packets. As the figure illustrates, far less packet drops are caused by malicious nodes during simulations with the trust based routing strategies, than with standard DSR. For standard DSR, approximately 47% of the packet drops are caused by malicious nodes. The lowest result, which is achieved with routing strategy 2 and 5 are approximately 17%, which results in a difference of 30%. Due to the possibility of other causes than malicious drops, a linear relation between the throughput and the percentage of malicious drops cannot be expected.
These results gives a better indication, than the results based on throughput, of the ability of the trust based strategies to avoid malicious nodes and supports the conclusion drawn in section 6.7.
There are still some uncertainties related to the results because it cannot be determined how many packets that were dropped internally.

6.9 Examining the Route Cache

The results presented in section 6.7 showed, by the use of trust values, that nodes were able to identify malicious nodes. These results were supported by the results from section 6.8 that showed that fewer packets were dropped by malicious nodes when trust based routing was used. To dig deeper into the trust based route selection the route selection it self is examined. One observation mentioned in section 3.1.1, is that once a route to a destination exist, the node will not actively try to obtain new routes, before a ROUTE ERROR indicating that the route is broken, is received. This might result in situations where a node is forced to use a route with malicious nodes. The extent of this problem is examined by gathering information about all routes that were available to a destination and the actual route chosen. Based on these results a metric that expresses how well the strategy selects routes is presented. The route information is written to a file at run time and preprocessed using a parser. The code for this parser is included in appendix P.2. The results where created during the same simulations that the results from section 6.8 and therefore the same parameters from Table 6-11 were used. However, only results for 28% percent malicious nodes are presented. As mentioned is section 4.1.7, the same method, `findRoutes()`, is used for finding routes for a packet and for finding routes to return as reply to ROUTE REQUEST's. To avoid calls to `findRoute()` that occurs due to ROUTE REQUEST's it is chosen to set the Reply_from_cache_on propagation flag to false. The results are presented in Figure 6-11.

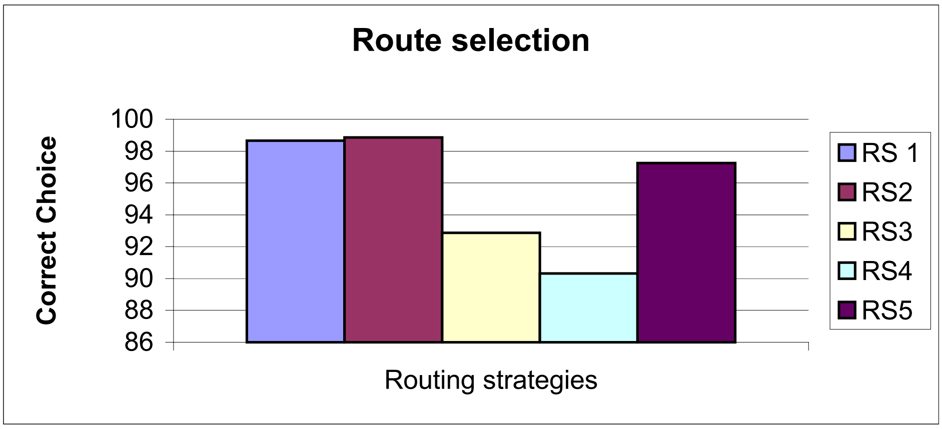


Figure 6-11: Comparison of routing strategies based on observation of the cache

The y-axis of the graph shows values calculated by Equation 6-3.

$$C = \sum_{1}^{n} \frac{g * 100}{g + gd} \qquad \text{Equation 6-3}$$

C: Correct choice
Where g: Number of good routes selected
gd: Number of good routes discarded
n: number of simulations

The total number of possible good routes that could be selected is the sum of the number of good routes that were selected and the number of good routes that were discarded. These results are based on 5 simulations with each strategy (resulting in *n* = 5). The formula expresses the percentage of correct routing choices that were made. The results show that routing strategy 1 and 2 outperforms the others with approximately 98% good routing choices. Strategy 4 has the lowest results of 90% good routing choices. These results show that the trust based route selection has a high percentage of correct route selections.

To examine causes for the relatively high percentage of malicious drops compared to the above results, it has been investigated how often a node is forced to choose a route containing malicious nodes. These results are presented in Figure 6-12.

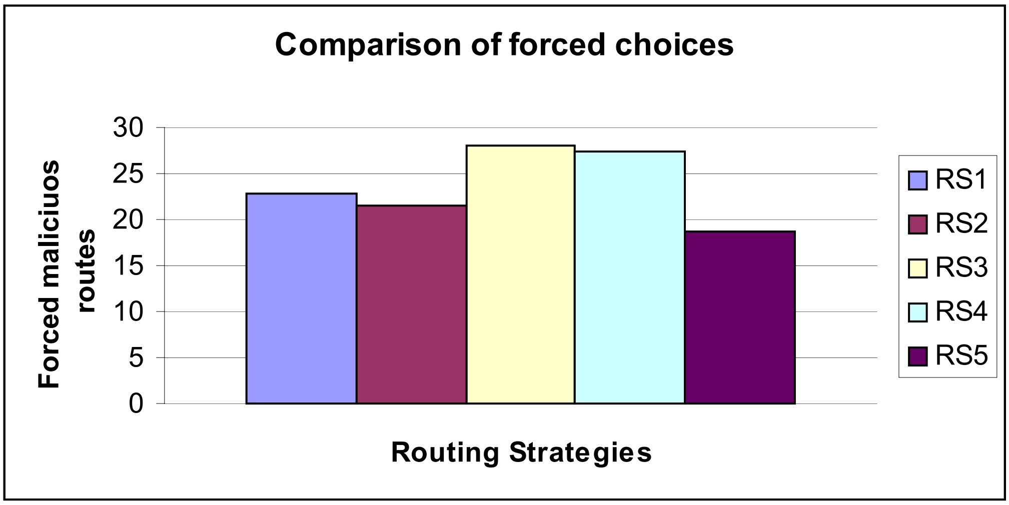


Figure 6-12: Graph showing how many times a route containing malicious nodes were the only choice.

The y-value of the graph was calculated using Equation 6-4.

$$F = \sum_{1}^{n} \frac{fb * 100}{t} \qquad \text{Equation 6-4}$$

F: Percentage of total route choices that were forced to be malicious.
Where fb: Number of bad routes selected with no alternative
t: Total number of routes selected

The equation expresses how many times a route with malicious nodes was selected because no alternative route was present. As the graph shows between 18% and 27% of the routes with malicious nodes where selected because no alternative route were available.

This result indicates that there is a bottleneck for trust based routing when only one route is available. This corresponds to the observation described in section 3.1.1. Unfortunaltly, time did not allow further investigation of this area.

6.10 Uncertainties

Due to the large number of variable parameters and the generated output some uncertainties are related to the results. In this section some of these uncertainties will be discussed.

The results from section 6.5.3 shows that the deviation in the results obtained from simulating with different scenarios could be more than 20%. Even though the most pessimistic scenario was used, an even worse might exist. The opposite is also the case, as another scenario could lead to better result than the ones achieved with the best scenario here.

The scenarios only consisted of 25 nodes moving with a maximum speed of 1 m/s. Fewer nodes would mean less possible routes and in such situations the trust based route selection strategies might not have the same ability avoid malicious nodes.

Using more nodes would mean more possible routes, which could give trust based routing a greater advantage.

All the parameters were estimated with one scenario and it cannot be excluded that other scenarios could have resulted in different values. And at the same time it cannot be ruled out that one trust based routing strategy performs better than others in certain unforeseen situations.

Only one set of parameters values were used for simulations of standard DSR and whether better or worse results could have been achieved with other values is not investigated.

To minimize these uncertainties a larger number of different scenarios should be used for the simulations, which would lead to a better statistical foundation. This is however a time consuming task, since one simulation with the used simulation platform takes approximately 8 minutes, which leads to a minimum of 3 hours to perform simulations for one routing strategy.

However, it can be stated with 90% certainty, that trust based routing achieves a higher throughput than DSR under the circumstances that have been simulated.

6.11 Summary

This section presents a summary of the simulations and derived results that have been discussed in chapter 6. Preliminary simulations have been carried out to determine

- The impact of using different simulation scenarios. The results showed that as much as 20% deviation between the highest and lowest achieved throughput was experienced. Based on these results the worst-case scenario was used for the remaining simulations.

- The Acknowledgement time out value was estimated to 0.07 seconds

- The initial trust value was estimated to –0.4

These values were used for the further simulations. Besides the preliminary simulations, several other simulations were described and the results analyzed. This has lead to the following results:

- Both standard DSR and DSR with trust based routing managed to deliver 72% of the packets with no malicious nodes in the network. This result can to some extent be explained by some integration problems in the existing Ns-2 implementation of DSR.

- All trust based route selection strategies outperformed standard DSR when malicious nodes were present in the network. The best result was a 15% increase in throughput, which was achieved with route selection strategy 1. The throughput for strategy 2 and 5 were close to that of strategy 1, where the throughput for strategy 3 and 4 were a bit smaller.

- Results based on nodes trust values and on the percentage of dropped packets that were dropped by malicious nodes, clearly indicated that malicious nodes where identified.

- Simulations that generated output directly from the route cache were presented. A metric that describes how good a strategy was to select good routes over routes with malicious nodes were used. The results from these simulation revealed that route strategy 1 and 2 selected the best possible route 98% times. Routing strategy 4 performed the worse with a correct choice 90% of the times. From the same simulation results, knowledge about how many times a route with malicious nodes where the only possible route, where deduced. These results showed that 18%- 25% of the times a route with malicious nodes were selected no alternative route was available.

7 Future Work, Improvements and Perspective

Time has, as always been a limiting factor, and several areas and solutions has been examined in thought but has not been put into practice. This chapter is related to the future work that could be performed to investigate interesting areas or lead to clarifications or improvements of the existing work. Furthermore, a perspective of the achieved results is presented. As the preceding chapter has indicated, the combinations of the trust related parameters and DSR parameters are almost infinitive. Whether a perfect combination exists and is still undiscovered and uncertain. Since the architecture for forming and updating trust is designed and implemented, it could be used for other purposes than just route selection. However, the area of using trust relationships, established in one content, in others is still under investigation, an it is not clear how, and if, trust can be transferred from one content to another. Therefore only one area for using the existing trust values for other purposes are proposed.

7.1 Introduction of grudging behavior

The CONFIDANT protocol, described in section 2.3.5, uses a mechanism to detect malicious behavior and punish these. If a node identifies another node as being malicious it starts to bear a grudge against that node. The grudge is put into practice as the node stops forwarding packets for the malicious node. The idea that malicious behavior has a negative consequence corresponds well to the perception that people has when other people acts malicious. The protocol has the flaw that once a node is identified as malicious, there is nothing that can change its evaluation. The trust based system that has been developed here adjust the trust that one node has in other nodes based on the events experienced. This means that malicious nodes that start to act according to protocol have their trust ratings increased, which seams to correspond better to real life situations where trust can broken, but also re-established. Therefore, the existing trust components could be used to apply grudging behavior to the protocol.

7.2 Using a sliding window mechanism for acknowledgements

With the current implementation acknowledgements for all data packets are requested. This results in an increased packet overhead, which it will be beneficial to decrease. As mentioned in section 4.1.5, a sliding window mechanism could be used to minimize the number of requested acknowledgement, which would decrease the routing overhead.

7.3 Derivation of knowledge by examining received packets

By analyzing the received packets and their source routes much information can be derived. One problem with the current system is that nodes are punished collective. This means that good nodes can be identified as malicious if they are on routes with malicious nodes. The evaluation of packet drops can be made more sophisticated. Figure 7-1 illustrates a situation that could be used to make a more sophisticated adjustment of trust values. A first receives a packet from C that has been successfully forwarded by B. This means that A increases its trust in B. A subsequently sends data to C but does not receive any acknowledgement. This leads to a strong indication that C is malicious because B just proved trustworthy. The current result is that both B and

C have their trust values decreased, but the gained knowledge could be used to only adjust C's value. By recognizing such situations the trust updating could be refined.

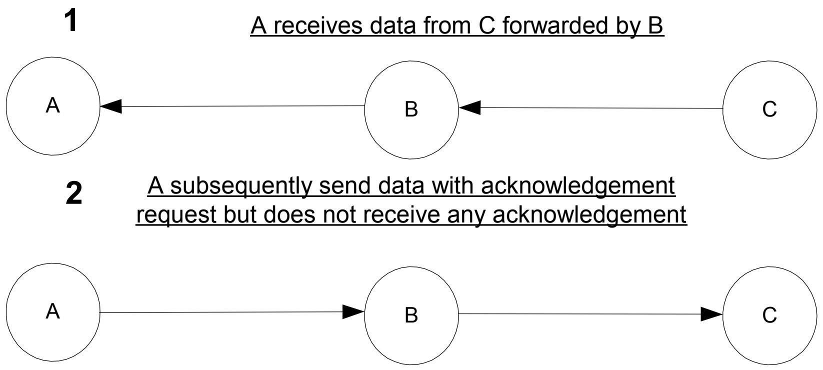

1
A receives data from C forwarded by B
A
B
C
2
A subsequently send data with acknowledgement request but does not receive any acknowledgement
A
B
C

Figure 7-1: A situation where A can derive information from the received packets

7.4 Examining cause and location of packet drops

As the analysis of the simulation results revealed, the throughput is not only affected by packets dropped by malicious nodes, but also by other factor such as the ARP implementation. The results indicate that several packets are dropped internally in the node because no route is found. Therefore it is recommended that further simulations should be carried out with particular emphasis on examining where and under which circumstances packets are dropped. Making further differentiates between the causes for packet drop, such as broken routes, queue overruns or missing routes, will make it far much easier to evaluate whether or how, the drops can be avoided.
Based on the knowledge gained with DSR and the Ns-2 implementation, especially the packet buffer in the DSRAgent, the ARP implementation and the methods for adding and deleting routes from the route cache should be examined. Furthermore, it should be investigated how nodes can request more routes when only one route with possible malicious nodes is available.

7.5 Decrease trust over time

Dependent on the size of the scenario, many nodes can be encountered. Some nodes might be encountered at some point in time, leave the scenario and then never again be encountered again. Therefore, the storage of trust information for all encountered nodes could lead to storage of a lot of stale information. The trust update function from Equation 4-2 involves inflation constant, *d*, which means that the trust value is automatically decreased based on this inflation constant. The inflation is based on events and not on time, meaning that until an event occurs no adjustment of the trust value occurs. By including some time based adjustment mechanism trust values could be disposed if they had not been used within some timeframe.

7.6 Perspective

This section present a perspective of the application of trust based routing.

By applying trust based routing to DSR a higher throughput was achieved when malicious nodes were present in the network. The effect on throughput of trust based routing when no malicious node was present seems insignificant. This means that if a higher throughput is required trust based routing can be recommended.

The increase in throughput does however have a cost on other areas. The mechanism for evaluating routes is more complicated and requires more computations than the one used by standard DSR. The increased computations will lead to a higher power consumption, which can present a problem for mobile devices, and it also means that the application of trust based routing results in a higher latency than that of standard DSR. Furthermore, trust values and acknowledgement information is stored during run time which requires more storage capacity of the device.

Furthermore, trust based routing makes use of acknowledgements to adjust trust values. This increases transmissions, which increase both routing overhead and power consumption.

The application of trust based routing have only focused on detection of malicious behavior that is expressed by nodes not forwarding packets and it does not cover other areas such as authentication and integrity.

Others, such as the CONFIDANT and the Pathrater - Watchdog mechanism have achieved better results on throughput with their methods than the results achieved with trust based routing. These approaches does however use different methods for identifying malicious nodes which presents some problems, such as collusion detection, that are not experienced with trust based routing.

7.7 Summary

Based on observations and gained knowledge from working with DSR and Ns-2 the future improvements and areas for further investigation listed below have been proposed:

- Using the existing trust values to let nodes bear grudges against nodes identified as malicious.

- Using a sliding window mechanism to limit the number of acknowledgement request.
- Creating more sophisticated mechanisms, based on situation recognition, for trust updating.

- Examining where and under which circumstances packets are dropped by non-malicious nodes.

Finally, a perspective of trust based routing and its advantages and disadvantages were presented.

8 Conclusion

This chapter presents the key contributions and conclusion of the M.Sc. Thesis work that has been carried out in relation to the project ”*Secure Routing in Mobile Ad Hoc Networks*”. The project has been carried out in the period 1st of July to 31st of December 2003 at the Department of Informatics and Mathematics Modelling, at the Technical University of Denmark, DTU.

The project has focused on routing in mobile wireless ad hoc networks, which are networks where nodes move around and establish connections with no fixed infrastructure.
The aim of the project has been to design and implement a system that can be used for trust based routing.

The stated primary objective is presented below.

> ***Primary objective****: To apply trust based route selection to the Dynamic Source Routing (DSR) protocol, in order to fortify the protocol and improve route selection, which can increase throughput in situations where malicious, nodes are present in the network.*

In order to accomplish the primary objective several areas has been examined.

The preliminary objectives for the project have been to examine ad hoc routing protocols, security in ad hoc networks, trust management systems and the concept of trust. In order to meet these objectives the following has been covered.

- The four ad hoc routing protocols DSDV, TORA, DSR and AODV have been examined with special emphasis on the DSR protocol.

- Several security mechanism, that can be used to establish different levels of security in ad hoc networks have been covered. This investigation revealed that obtaining security properties such as confidentiality and integrity is possible, but due to the lack of an existing infra structure in the network, also difficult. Further, several methods for increasing throughput in ad hoc networks where malicious nodes refuse to forward packets were described.

- Trust as a concept has been studied and several frameworks that can be used to express and use trust in a formalized way has been accentuated. Furthermore, the area of trust management systems has been investigated.

In main objectives have involved analysis of the DSR protocol, design and implementation of a system that incorporates trust based route selection. They have been met by covering the following:

- In order to identify weaknesses and situations where trust could be incorporated to fortify the DSR protocol, the DSR protocol was analyzed. This analysis revealed several possible vulnerabilities and identified how trust based route selection can be applied to DSR.

- A mechanism that establish and updates trust relationships are designed. This mechanism is used to apply trust based route selection to the DSR protocol, to fortify the protocol against the presence of malicious nodes in the network. The trust based routing is achieved by experimenting with five different routing strategies that have been designed and implemented.

- The designed system is implemented in C++ and the components are integrated with an existing DSR implementation in the network simulator Ns-2.

To meet the post objective of analyzing the impact that trust based route selection has and identify areas of improvement, several simulations with the implemented systems has been carried out using the Ns-2 simulator. The simulations have provided empirically data about the protocols behavior, which has been analyzed.

The results show that DSR fortified with trust based routing achieves a higher throughput than standard DSR when malicious nodes are present in the network. The best results are achieved with a trust based routing strategy that used the average of the nodes on the routes trust values. With this strategy a 15 % increase in throughput, compared to standard DSR, was achieved. Moreover, the results clearly indicate that malicious nodes are identified, which is supported by the fact that far less drops are caused by malicious nodes. This result corresponds well to results presented by John Keane.

To investigate how well the trust based routing strategies selects routes without malicious nodes, further simulations was conducted. The results show that in as much as 98% of the times, a correct choice is made. These results also reveals that as much as 28% of the routes that contained malicious nodes, is selected because no alternative route is present.

Several proposals to improvements and areas of further investigation are suggested and described. These are listed below:

- Introduction of grudging behavior
- Applying a sliding window mechanism for acknowledgements
- Deriving information from received packets
- Examine cause and location for packet drops further
- Decrease of trust over time

The key contributions to the area of secure routing in ad hoc networks have been:

- Design, application and evaluation of different trust based routing strategies.

- Evaluation of trust based routing with emphasis on: Ability of the to detect malicious nodes, select correct routes and causes for route selection.

The overall conclusion is that the stated aim has been fulfilled.

Concerning the objectives the conclusion is, that the presented contribution and work fulfills the stated objectives.

Finally, it is my conclusion, based on the results that others have achieved with the DSR protocol and the results produced during this project, that the trust based routing mechanism can be refined which can lead to improvement of the results presented in this M.Sc. Thesis.

In order to achieve improvements I recommend that focus be placed in the area of requesting routes when only one route with possible malicious nodes is available in the cache.

A Bibliography

[Anderson] Ross Anderson, Frank Stajano: *The Resurrecting Duckling: Security Issues for Ad Hoc Networks*, In Proceedings of the 7th International Security Protocols Workshop page 172-194, 1999

[Bajaj] L. Bajaj, M. Takai, R. Ahuja, K. Tang, R. Bagrodia, and M. Gerla: *GloMoSim: A Scalable Network Simulation Environment*, Technical Report 990027, UCLA Computer Science Department, 1999

[Bertsekas1] Dimitri P. Bertsekas, Robert Gallager: *Data Networks (2nd Edition)*, Prentice Hall, 1991

[Blaze1] Matt Blaze, Joan Feigenbaum, Jack Lacy: *Decentralized Trust Management*, Proceedings IEEE Conference on Security and Privacy, Oakland CA, 1996

[Blaze2] Matt Blaze: *Using the KeyNote Trust Management System*, http://www.crypto.com/trustmgt/kn.html, November 1999, Updated 1 March 2001, last visited November 2003.

[Blaze3] Matt Blaze, Joan Feigenbaum, Angelos D. Keromytis: KeyNote: Trust Management for Public-key Infrastructures, In Proc. Cambridge 1998 Security Protocols International Workshop, pages 59--63, 1998.

[Broch1] Josh Broch, David B Johnson, David A Maltz, Yih-Chun Hu, Jorjeta Jetcheva: *A Performance Comparison of Multihop Wireless Ad Hoc Networking Protocols*, Proceeding of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom98), 1998

[Buchegger1] Sonja Buchegger, Jean-Yves Le Boudec: *Nodes Bearing Grudges: Toward Routing Security, Fairness and Robustness in Mobile Ad Hoc Networks*, in 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, January 2002

[Buchegger2] Sonja Buchegger, Jean-Yves Le Boudec: *Performance Analysis of the CONFIDANT Protocol Cooperation Of Nodes – Fairness in Dynamic Ad-Hoc Networks*, In Proceedings of IEEE/ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC), Lausanne, Schwitzerland, June 2002

[Buttyan] Levente Buttyan, Jean-Pierre Hubaux, *Nuglets: a Virtual Currency to Stimulate Cooperation in Self-organized Mobile Ad hoc Networks*, Technical Report DSC/2001, 2001

[Corson1] M. Scott Corson, Anthony Ephremides: *A Distributed Routing Algorithm for Mobile Wireless Networks*, Journal of ACM/Baltzer Wireless Networks Vol. 1 no.1, 1995

[Desmedt] Y Desmedt: *Treshold Cryptography*, European Transactions on Telecommunication, 1994

[Fall] Kevin Fall, Kannan Varadhan: *The ns Manual (formerly ns Notes and Documentation)*, http://www.isi.edu/nsnam/ns/doc/index.html, last visited 11-12-2003

[Gambetta] Diego Gambetta, *Can we trust trust?*, Trust, Blackwell, Page 217, 1990

[Gamma] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (The Gang Of Four): *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994

[Glom] GloMoSim website, http://pcl.cs.ucla.edu/projects/GloMoSim/, last visited 12-12-2003

[Greis] Marc Greis: *Tutorial for the simulator "ns"*, http://www.isi.edu/nsnam/ns/tutorial/ns.html, last visited December 2003

[Guoyou1] Guoyou Ho: *Destination-Sequenced Distance Vector (DSDV) Protocol,*

[Heesch] Dimitri van Heesch creator of Doxygen, Doxygen website : http://www.stack.nl/~dimitri/doxygen/, last visisted 30-12-2003

[Hu] Yih-Chun Hu, David B. Johnson: *Ensuring Cache Freshness in On-Demand Ad Hoc Routing Protocols*, POMC'02, 2002

[Jain] Raj Jain : *The art of computer systems performance analysis, techniques for experimental design, measurement, simulation*, *and modelling*, John Wiley and Sons Inc, 1991

[Johnson1] David B Johnson, David A Maltz, Josh Broch: DSR: *The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks*, 1994

[Johnson2] David B Johnson, David A Maltz, Yih-Chun Hu, *The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks (DSR)*, RFC Internet Draft, 2003

[Jonker] Catholijn M. Jonker, Jan Treur: *Formal Analysis of Models for the Dynamics of Trust based on Experience*, Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering ({MAAMAW}-99)", volume 1647, pages,221--231, Springer-Verlag: Heidelberg, Germany",1999

[Keane] John Keane: *Trust based Source Routing in Mobile Ad Hoc Networks*, dissertation submitted to the university of Dublin, 2002

[Kohl] John Kohl, B. Clifford Neuman: *The kerberos network authentication service (V5)*, Request for Comments (Proposed Standard) RFC 1510, Internet Engineering Task Force, September, 1993.

[Marsh] Stephen Paul Marsh, *Formalizing Trust as a Computational Concept*, Ph.D. Thesis, Department of Mathematics and Computer Science, University of Stirling, 1994

[Marti] Sergio Marti, T.J Giuli, Kevin Lai, Mary Baker: *Mitigating Routing Misbehaviour in Mobile Ad Hoc Networks,* Mobile Computing and Networking pages 255-265, 2000

[Mcknight] D. Harrison Mcknight, Norman L. Chervany: *The Meaning of Trust*, Working paper, Carlson School of Management, University of Minnesota, 1996

[NS2] S. McCanne, S. Floyd: *ns--Network Simulator*, http://www-mash.cs.berkeley.edu/ns/.

[NS2-M1] NS-2 mail archive http://mailman.isi.edu/pipermail/ns-users/2003-May/031977.html, last visited 12-12-2003

[NS2-M2] NS-2 mail archive http://mailman.isi.edu/pipermail/ns-users/2003-November/037234.html, last visited 17-12-2003

[NSEx] *Ns by Example*, http://nile.wpi.edu/NS/, last visited December 2003

[Park1] Vincent D. Park, M. Scott Corson: *A Higly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks*, published in the proceedings of INFOCOM 97, 1997

[Parsec] Parsec language website: http://pcl.cs.ucla.edu/projects/parsec/, last visited 27-12-2003

[Perkins1] Charles E. Perkins, Elizabeth M. Royer: *Ad-hoc On-Demand Distance Vector Routing*, in MILCOM '97 panel on Ad Hoc Networks, Nov. 1997

[Perkins2] Charles E. Perkins, *Ad Hoc Networking*, Addison Wesley Professional, ISBN: 0-201-30976-9, chapter 8 ,2001

[Seigneur] J. Seigneur, S. Farrell and C. Damsgaard Jensen, E. Gray, Y. Chen: *End-to-end Trust Starts with Recognition*, Proceedings of the First International Conference on Security in Pervasive Computing, 2003

[Sharp] Robin Sharp: *Principles of Protocol Design Draft Second Edition*, DTU-TRYK, Technical University of Denmark, DTU, 2002

[Siek1] Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine: *The Boost Graph Library: User Guide and Reference Manual* Addison-Wesley, ISBN 0-201-72914-8, 2002

[Weeks] Stephen Weeks: *Understanding Trust Management Systems*, In IEEE Symposium on Security and Privacy, 2001

[Yang] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, Martin Strauss: *REFEREE: Trust Management for Web Applications*, Computer Networks and ISDN Systems Vol 29, 1997

[Yi] Seung Yi, Prasad Naldurg, Robin Kravets: *Security-Aware Ad Hoc Routing for Wireless Networks*, Proceedings of the 2001 ACM International Symposium on Mobile ad hoc networking & computing, 2001

[Zhou] Lidong Zhou, Zygmunt J. Haas: Securing Ad Hoc Networks, IEEE Network Magazine, vol. 13, no.6, November/December 1999

[Zou] Xukai Zou, Byrav Ramamurthy, Spyros Magliveras: *Routing Techniques in Wireless Ad Hoc Networks – Classification and Comparison*, Proceedings of the Sixth World Multiconference on Systemics, Cybernetics, and Informatics--SCI, July 2002

B List of used terms and definitions

Ad Hoc network	Short for Mobile Wireless Ad Hoc network.
Ad hoc routing protocol	A routing protocol designed and aimed especially at Mobile Wireless Ad Hoc networks.
Collaborative Ad hoc Network	An ad hoc network where the nodes does not have any common task (other than to have their own packets forwarded) to achieve by forming the network.
Hop-by-hop routing	Routing where only the next hop towards the destination is known.
Initiator	The node that is the source of a message.
Known node	A node that has been encountered during communication.
Malicious node	A node that does not act according to protocol, for instance by dropping packets, forge packets etc.
Message	A specific protocol packet.
Mobile device	A device, such as cell phones, PDA's or laptops that operates on battery power and can be carried around.
Mobile Wireless Ad Hoc network	A network made op of mobile devices that communicates via wireless communication.
Node	A mobile device that is being held by a person or integrated into another device.
Packet	Data that is being transmitted between to nodes. It can be application data or routing related data.
Proactive/table driven protocol	A routing protocol that periodically transmits routing information.
Reactive/On-demand	A routing protocol that only transmits routing information on demand, when a route is needed.
Source routing	Routing where the sender includes the entire route to the destination in the packet header.
Throughput	The number of received application packets divided by the number of send application packets.
Trusted server	A server that can be trusted to distribute cryptographic keys, certificates etc, and handle authorization.
Unknown node	A node that has not been encountered during communication.

C Nomenclature

$\bar{x}$	Mean
$z_{1-\alpha/2}$	$(1-\alpha/2)$ interval
C	Correct choice
Con	Confidence interval
d	A constant used to express the inflation of trust
E	A partially ordered set of experience classes for
ES	The set E^N of experience sequences
ev	The experience
F	Percentage of total route choices that were forced to be malicious
fb	Number of bad routes selected with no alternative
g	Number of good routes selected
gd	Number of good routes discarded
N	The set of natural numbers
n	Sample size/ *number of simulations*
S	Standard deviation
T	A partially ordered set of trust qualifications/trust values
t	Total number of routes selected
tc	Time out constant
TO	Total time out value
tv	The existing trust value

D List of used acronyms

AODV	Ad Hoc On Demand Distance Vector
AP	Authentication Process
APER	A Peer Entity Recognition scheme
API	Application Programmer’s Interface
ARP	Address Resolution Protocol
CBR	Constant Bit Rate
DSR	Dynamic Source Routing
DSVD	Destination-Sequenced Distance Vector
ER	Entity Recognition
MAC	Medium Access Control
NS	Network Simulator
PKI	Public Key Infrastructure
SAR	Security Aware Routing
STL	Standard Template Library
TORA	Temporally Ordered Routing Algorithm
UML	Unified Modelling Language

D - List of used acronyms

E Appendices (F – P)

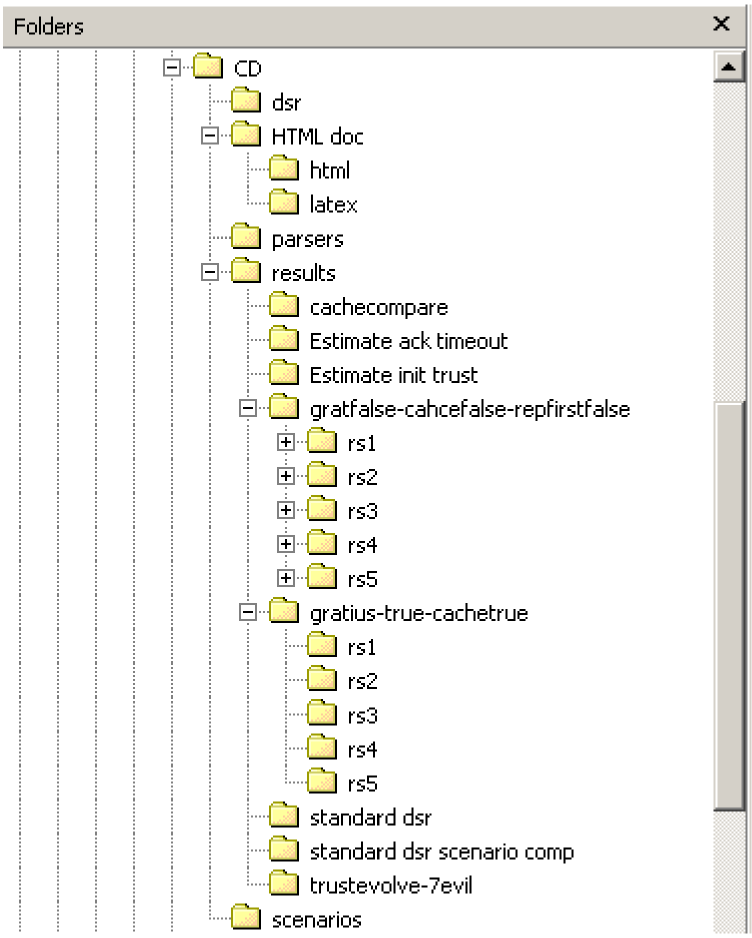
F List of figures

G List of tables

H List of equations

I Content of the CD

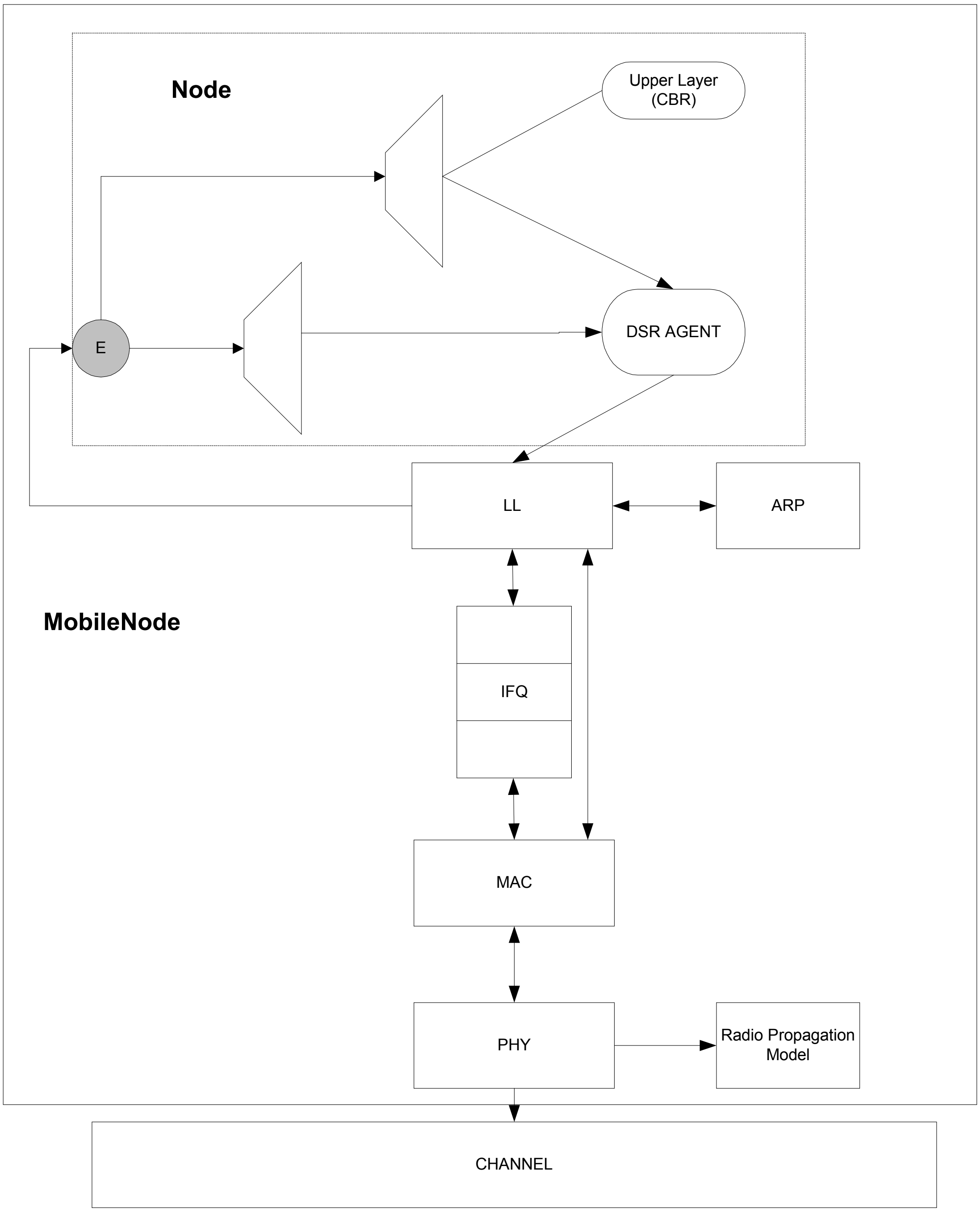
Enclosed with this thesis is a CD containing different material. The screenshot shows the folder structure of the CD. A readme file with a brief description of the material is included on the CD.



```
Folders
CD
    dsr
    HTML doc
        html
        latex
    parsers
    results
        cachecompare
        Estimate ack timeout
        Estimate init trust
        gratfalse-cahcefalse-repfirstfalse
            rs1
            rs2
            rs3
            rs4
            rs5
        gratius-true-cachetrue
            rs1
            rs2
            rs3
            rs4
            rs5
        standard dsr
        standard dsr scenario comp
        trustevolve-7evil
    scenarios
```

Appendix Figure 1: Directory structure of enclosed CD

J Structure of mobile node in Ns-2



C: Classifier LL: Link Layer IFQ: Interface
Queue ARP: Address Resolution MAC: Medium Access Control PHY : Network Interface

Appendix Figure 2

The structure of the mobile node implementation in Ns-2 is quite complex and is described to some details in the Ns manual [Fall]. Because simulations results have turned out to be influenced by this implementation an ultra brief description is given here.

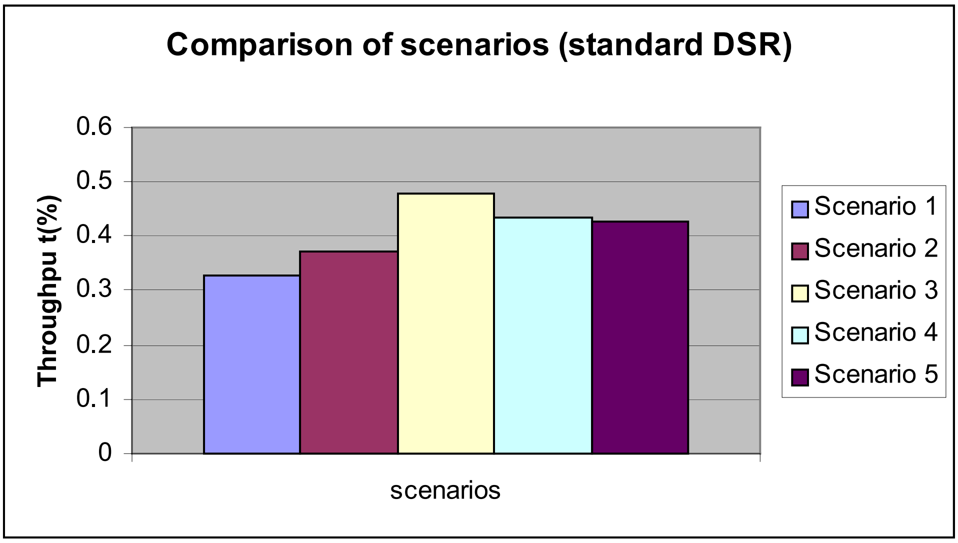
As seen the `DSRAgent` is only one part of the mobile node.

The link-layer is responsible for simulating the data link protocols. An important task for the link layer is adding a MAC destination address in the MAC header of the packet.

The link-layer is connected to an Address Resolution Protocol (ARP) module. The ARP is responsible for finding the hardware address that corresponds to a packets next destination. Once the hardware address of a packet's next hop is known, the packet is inserted into the interface queue.

To bind the link-layer and the MAC layer an interface queue is used. This queue gives priority to routing packets.
As seen both incoming and outgoing packets pass through the interface queue.

K Result from simulating DSR with different scenarios



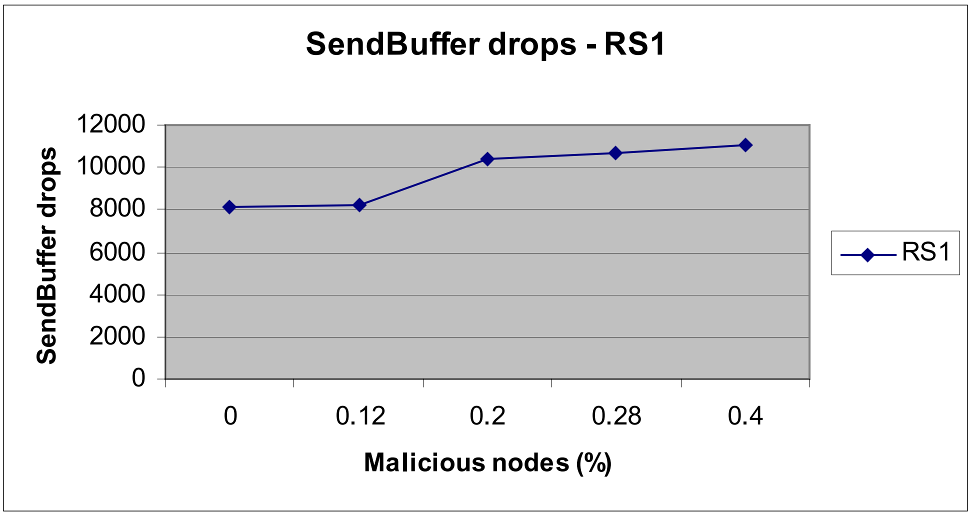
Appendix Figure 3

Seven malicious nodes where used for these simulations.

L Simulation platform

All simulations have been performed on a PC that used an Intel Pentium III 1150 MHz processor and had 256 MB RAM. The operating system was a Linux Red Hat version 7.2. This equipment was used because it was what DTU provided. Ns-2 version 2.18b was used for simulations and the g++ compiler version 2.96 was used to compile Ns-2 and the trust related extension.

M Sendbuffer drops during simulation with RS1



Appendix Figure 4

The figure presents the number of packets that are dropped internally from the sendbuffer. Packets are dropped for two reasons: The buffer is full or no route is found within a specified timeframe.

N Output from simulations

While a simulation is running, output is written to a trace file. This trace file holds information about sent, received, forwarded or dropped packets. It is possible to specify the levels of trace that one wish to trace, for instance MAC level or Routing level. To give the reader an idea of the level of details that are contained in the trace file, the figure shows to lines from the trace file.

```
r 7.550313772 _8_ AGT  --- 16 cbr 556 [a2 8 7 800] ------- [7:1 8:1 32 8] [5] 1 1
s 7.770552227 _7_ AGT  --- 18 cbr 512 [0 0 0 0] ------- [7:1 8:1 32 0] [6] 0 1
```

Appendix Figure 5 : Two lines from a trace file

The first line describes a packet being received at time 7.550313772 at agent 8. The packet is a CBR packet. The trace file also contains information about the route the packet was sent by, IP headers and packet size. Several explanations of the trace file format exist on the Internet [Fall]. In order to register malicious drops and other self defined events the method `exttrace()` has been implemented and used to write to the trace file.

O Implementation details

O.1 Scanning for timed out acknowledgements

The method `scanForOldACKs()` is used to examine the map that contains all transmitted acknowledgement requests. At this point this method is called every time a new acknowledgement is requested. This ensures that a bad route is never picked because the trust in the nodes has not been updated due to a timed out acknowledgement. This can however also result in unessesary computations. The minimum time out value for an acknowledgement is in the case where the route is of length two, meaning that the node is a neighbour. This value can be determined by Equation 4-6. If a node is generating a lot of traffic it might be beneficial to run the `scanForOldACKs()` method in a separate thread that is activated according to the minimum timeout value determined by Equation 4-6. It should be noticed that acknowledgements are required from neighbours due to the malicious behavior described in section 3.3 were malicious nodes does not return acknowledgements. .

O.2 Malicious behavior

When a `DSRAgent` is created from the OTcl script it is not assigned an id in its constructor. This first happens when the `command()` method is called. These ids are integers between 0 and the number of nodes minus one. The ids are used to determine whether or not a node should act as malicious. In the current implementation the malicious nodes are hard coded in the method `initEvilNodes()`. If the id of a node is contained in an array in the method `initEvilNodes()`, true will be returned and stored in a class variable. Ideally, the ids for the nodes that should carry out malicious behavior should be provided through the OTcl script so compilation of C++ code could be avoided. Several examples exist on how to bind a method or a variable from OTcl to C++ [Fall] but none of them seems to work for mobile nodes because mobile nodes in Ns-2 consist of several different objects.

O.3 File format

The content of the TrustManager can be saved after each simulation by calling the `save()` method. For easiness the content is saved in a binary format. This makes it easy to read from a programming point of view but difficult to read with human eyes. Therefore the EBNF for the format is presented in Appendix Figure 6 here in case anyone should want to port it to other applications.

```
<Nodes>::= {<Node>}
<Node>::<Nodeid><TrustValue><NrOfExperiences>{<Experience>}
<Nodeid>::= integer
<TrustValue>::= double
<NrOfExperiences>::= integer
<Experience>::= double
```

Appendix Figure 6: EBNF describing the file format for saved trust information

P Source code

P.1 DSRParser.java

```
import java.io.*;
import java.util.StringTokenizer;
//By Lennart Conrad
//Used to parse Trace files
public class DSRParser
{

        public DSRParser(){}


        /*
        **      THe methods reads from the file given as argument
        **      Only the first parameter in every line is parsed- the rest is ignored
        */
        public void read(String filename)
        {
                try
                {
                        FileReader in = new FileReader(filename);
                        BufferedReader reader = new BufferedReader(in);
                        String line ="";
                        int i = 0;//nr of lines
                        int r =0;//received
                        int EDATA = 0;//evil drop of data
                        int ED = 0;//evil drop
                        int EAD = 0;//evil drop of ack
                        int D = 0;//drop old format
                        int d = 0;//new format
                        int Ssb = 0; //send buffer drop

                        int f = 0;//forward
                        int SF = 0;//forwarded in DSR forward


                        int AS= 0;//ack send
                        int SO=0;//DSR send
                        int s = 0;//send
                        int S = 0;//Send DSR

                        int others = 0;//others
                        int colerrors =0;//do we read more than the first col?

                        while(line != null)
                        {
                                line = reader.readLine();
                                if(line != null)
                                {
                                        StringTokenizer st = new StringTokenizer(line);
                                int col = 0;
                                                while (st.hasMoreTokens())
                                {
                                col++;//make sure we only read the first column
                                                        String value =st.nextToken();
                                if(value.equals("D"))//old format D
                                {
                                        //several drops can occur when the simulation
ends and these are not counted
                 D++;
                                                                while (st.hasMoreTokens())
                                                {
                                                        value =st.nextToken();

        if(value.equals("END"))//END of simulation
                                                                                {
                                                                                  D--
;//count 1 down

        break;
```

```
                                        }

                                }
                }
                    else if(value.equals("d"))//new format d
                {
                        d++;
                }
                            else if(value.equals("EAD"))//new
format d
                {
                        EAD++;
                }
                            else if(value.equals("EDATA"))
                    {
                        EDATA++;
                    }

                            else if(value.equals("ED"))
                    {
                        ED++;
                    }
                            else if(value.equals("Ssb"))
                    {
                        Ssb++;
                    }

                            else if(value.equals("SO"))//new
format d
                {
                        SO++;
                }
                            else if(value.equals("AS"))
                    {
                        AS++;
                    }
                    else if(value.equals("s"))
                    {
                        s++;
                    }
                            else if(value.equals("S"))
                    {
                        S++;
                    }
                            else if(value.equals("r"))
                    {
                        r++;
                    }
                    else if(value.equals("f"))
                    {
                        f++;
                    }
                    else if(value.equals("SF"))
                    {
                        SF++;
                    }

                            else
                    {
                        break;//we only read the first column
                    }
                    //tjeck column
                            if(col > 1)
                            {
        colerrors++;
                            }
                    }
                    i++;//we read 1 more line
                }
            }
            System.out.println("Read: "+i+" lines");
            System.out.println("D (old format): "+D+" ");
            System.out.println("s: "+s+" ");
            System.out.println("r: "+r+" ");
            System.out.println("ED: "+ED+" ");
            System.out.println("EAD: "+EAD+" ");
            System.out.println("AS: "+AS+" ");
```

```
                        System.out.println("Ssb: "+Ssb+" ");
                        System.out.println("SO: "+SO+" ");
                        System.out.println("f: "+f+" ");
                        System.out.println("d (new format): "+d+" ");
                        System.out.println("EDATA: "+EDATA+" ");




                        //total send (DSR)
                        int ts = SO +AS+s;
                        System.out.println("Total number of send( SO+AS+s ): "+ ts);
            //total dropped
                        int td = EAD+d+D+ED+Ssb;
                        System.out.println("Total number of dropped( EAD+d+D+ed ): "+
td);
                  //total received
                        System.out.println("Total number of received( r ): "+ r);
                  //s -(r+d)
                        System.out.println("(Send - (dropped + received )): "+  (ts-
(td+r)) );
                  //total forwarded
                        int tf = f+SF;
                        System.out.println("Forwarded: (f +SF) "+ (tf) );


      int intr = td+r+ts+tf;
                        System.out.println("Read: "+i+" lines, and identified: " +(intr)
+ "interresting tokens");
                  System.out.println("Column errors: "+colerrors);

                        double percent = ((s-r)*100)/s;
                        System.out.println("Percentage dropped of send: "+percent+" ");
                }
                catch(IOException ie)
                {
                        System.out.println("Usage: DSRParser filename");
                        //System.out.println(ie.getMessage());
                }


        }
        public static void main(String args[])
        {
                System.out.println("DSRParser!");
                String file = args[0];
                DSRParser dsr = new DSRParser();
                dsr.read(file);
        }
}
```

P.2 RouteParser.java

```
import java.io.*;
import java.util.StringTokenizer;
import java.util.Stack;
import java.util.Iterator;

/*
**      By Lennart Conrad
**      used to parse information written from Mobicahce.
**      This is used to see which routes are picked, and when bad routes are picked
*/

public class RouteParser
{
  int NR_OF_NODES = 25;//specify according to scenario
        int NROFEVILNODES; cd ..//specify according to scenario
        Stack stack = new Stack();
        public RouteParser(int a)
        {
        NROFEVILNODES = a;
        }


        /*
        **      The methods reads from the file given as argument


        **      Only the first parameter in every line is parsed- the rest is ignored
        */
        public void read(String filename)
        {
               try
               {
                      FileReader in = new FileReader(filename);
                      BufferedReader reader = new BufferedReader(in);
                      String line ="";

      int linenumber = 0;
                      int ok = 0;
                      int notok = 0;
                      int good_discarded = 0;
                      int[] nodes= new int[NR_OF_NODES];
                      int[] discarded= new int[NR_OF_NODES];
                      String[] parsedline;
                      //Arrays.fill(nodes,0);
                      while(line != null)
                      {
                             line = reader.readLine();

                             if(line != null)
                             {
                                           parsedline = line.split(" ");
                                           StringTokenizer st = new
StringTokenizer(line);
                             int col = 0;
                                           while (st.hasMoreTokens())
                             {
                             String value =st.nextToken();
                             if(value.equals("NotOk:"))//
                             {
                                    String id =st.nextToken();
                                                         int nid =
Integer.parseInt(id);
                                                         nodes[nid] = nodes[nid]
+1;
                                    notok++;
                                                         //now go through the stack
and detrmine if good routes were discarded
                Iterator it = stack.iterator();
                                                         while(it.hasNext())
                                                         {
                                                                String[] l =
(String[])it.next();
                                                                boolean hasevil =
false;

        if(!l[1].equals(id))
                                                                {
                      break;
                                                                }
                                                                for(int e = 4;e
<l.length;e++)
                                                                {
                             if(l[e].equals("]"))//We are done with the nodes on the
route
                                                                               {

if(!hasevil)//this route actually was OK!

        {
                             discarded[nid]=discarded[nid]+1; //register who discards
good routes

               good_discarded++;

               //write the one used

               System.out.print("Used: ");

               for(int kk = 0;kk<parsedline.length;kk++)//see which we didn't use
               {
                               System.out.print(parsedline[kk]+" ");

               }

               System.out.println(" ");

               System.out.print("Discarded: ");

               //write the one we discarded

               for(int k = 0;k<l.length;k++)//see which we didn't use

               {
                               System.out.print(l[k]+" ");

               }

               System.out.println(" ");


       }

       break;
                                                                                    }
                                                                                    else
                                                                                    {

       int nodeid = Integer.parseInt(l[e]);

       if(isUsingEvil(nodeid))

       {

               hasevil = true;

       }
                                                                                    }
                                                                    }
                                                       }
                                                             //clear stack
                                                             stack.clear();
                                                     }
                                      else if(value.equals("Ok:"))//clear stack
                               {
                                              stack.clear();
                                                                    ok++;
                               }
                                                     else if(value.equals("Node:"))
                               {
                                              stack.push(parsedline);//save it

                                                     }

                                                     else
                                      {
                                              break;//we only read the first column
                                      }

                                      }

                               }
                               linenumber++;
                       }
                       System.out.println("Good routes picked: "+ok);
                       System.out.println("Bad routes picked: "+notok);
                       System.out.println("Total number good routes that was discarded: 
"+good_discarded);
                       System.out.println("Total number of routes: "+(notok+ok));
                       for(int j = 0;j <nodes.length;j++)
                       {
       System.out.println("Node: "+j+" picked: "+nodes[j]+ " bad routes and discarded: 
"+discarded[j]+"good ones");
                       }
```

```
                }
                catch(IOException ie)
                {
                        System.out.println("Usage: RouteParser -nrofevilnodes -
filename");
                        //System.out.println(ie.getMessage());
                }


        }

        boolean isUsingEvil(int id)
        {
        //evil nodes id are in this array
        //for now they are hardcoded but they should be read from a file
        int evilnodes[] = {1,2,7,10,16,19,23};//,13,24,5};//max of 10
  //rmnsaddr_t evilnodes[] = {14,12,20,18,11,3,19,22,13,5};
        for(int i = 0; i<NROFEVILNODES;i++)
        {
        if(id == evilnodes[i])
                {
                        return true;
                }
        }
  return false;
}



        public static void main(String args[])
        {
                System.out.println("RouteParser!");
                int nrofevil = Integer.parseInt(args[0]);
                System.out.println(nrofevil);
                String file = args[1];
                RouteParser parser = new RouteParser(nrofevil);
                parser.read(file);
        }
}
```

P.3 Otcl script

```
#LC tcl file for trust simulations with DSR

# ======================================================================
# Define options
# ======================================================================
set val(chan)           Channel/WirelessChannel    ;# channel type
set val(prop)           Propagation/TwoRayGround   ;# radio-propagation model
set val(netif)          Phy/WirelessPhy            ;# network interface type
set val(mac)            Mac/802_11                 ;# MAC type
set val(ifq)            Queue/DropTail/PriQueue    ;# interface queue type
set val(ll)             LL                         ;# link layer type
set val(ant)            Antenna/OmniAntenna        ;# antenna model
set val(ifqlen)         1000                         ;# max packet in ifq
set val(rp)             DSR                        ;# routing protocol
set val(seed)           1.0                        ;#

#LC remember to set nodes correct
# ---------------- simulation with 4 nodes, ex 1 ------------------
#set val(nn)              4 ;# number of mobilenodes
#set val(x)                                                  1000
;# X dimension of the topography
#set val(y)              1000                          ;# Y dimension of the
topography
#The cbr pattern is defined in this file and assiociated with cb
#set val(cp)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/4nodes/cbr-lc-4nodes";
#The scenario (nodes movement and connections) is defined in this file and assiociated
with sc
#set val(sc)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/4nodes/scen-lc-4nodes";
```

```
#LC remember to set nodes correct
# ------------------- simulation with 25 nodes ------------------------------
set val(nn)              25 ;# number of mobilenodes
set val(x)                                                  1000
;# X dimension of the topography
set val(y)              1000                    ;# Y dimension of the topography
#The cbr pattern is defined in this file and assiociated with cb
#20 connections
set val(cp)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/25nodes/cbr-lc-25";
#50 connections
#set val(cp)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/25nodes/cbr-lc-25-50con";

#The scenario (nodes movement and connections) is defined in this file and assiociated
with sc
#set val(sc)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/25nodes/scen-lc-25-ex1";
#set val(sc)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/25nodes/scen-lc-25-ex2";
#set val(sc)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/25nodes/scen-lc-25-ex3";
#set val(sc)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/25nodes/scen-lc-25-ex4";
#set val(sc)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/25nodes/scen-lc-25-ex5";

#slow movement 1 m/s pause 60s
set val(sc)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/25nodes/scen-lc-25-p60-m1.0";

#no movement
#set val(sc)             "/home/s973586/ns-allinone-2.1b8/ns-
2.1b8/tcl/ex/lcsims/25nodes/scen-lc-25-nomove";
set val(stop)           500.0                      ;# simulation time

# ======================================================================

Agent/Null set sport_         0
Agent/Null set dport_         0

Agent/CBR set sport_          0
Agent/CBR set dport_          0

# unity gain, omni-directional antennas
# set up the antennas to be centered in the node and 1.5 meters above it
Antenna/OmniAntenna set X_ 0
Antenna/OmniAntenna set Y_ 0
Antenna/OmniAntenna set Z_ 1.5
Antenna/OmniAntenna set Gt_ 1.0
Antenna/OmniAntenna set Gr_ 1.0

# the above parameters result in a nominal range of 250m
set nominal_range 250.0
set configured_range -1.0
set configured_raw_bitrate -1.0


# ======================================================================
# Main Program
# ======================================================================

#Create a simulator object
set ns_        [new Simulator]

#Open the trace file
set tracefd     [open lc-out-tdsr.tr w]
$ns_ trace-all $tracefd
#$ns_ use-newtrace

# set the new channel interface.
#set chan        [new $val(chan)]

#Open the nam file
set namtrace [open lcout.nam w]
$ns_ namtrace-all-wireless $namtrace 1000 1000
```

```
#Set up topography object to keep track of movement of nodes
set topo       [new Topography]

#Provide topography object with coordinates
$topo load_flatgrid $val(x) $val(y)


# Create God
create-god $val(nn)

#Configure the nodes
        $ns_ node-config -adhocRouting $val(rp) \
                       -llType $val(ll) \
                       -macType $val(mac) \
                       -ifqType $val(ifq) \
                       -ifqLen $val(ifqlen) \
                       -antType $val(ant) \
                       -propType $val(prop) \
                       -phyType $val(netif) \
                       -channelType $val(chan)\
                       -topoInstance $topo \
                       -agentTrace ON \
                       -routerTrace OFF \
                       -macTrace OFF \
                       -movementTrace ON
                         #-channel $chan


#Create the specified number of mobilenodes [$val(nn)] and "attach" them
#to the channel.


        for {set i 0} {$i < $val(nn) } {incr i} {
        puts "i: $i"
        set node_($i) [$ns_ node]

                $node_($i) random-motion 0           ;# disable random motion
        }

#Define node movement model


puts "Loading connection pattern..."
source $val(cp)

#Define traffic model
puts "Loading scenario file..."
source $val(sc)

# Define node initial position in nam
for {set i 0} {$i < $val(nn)} {incr i} {

    # 15 defines the node size in nam, must adjust it according to your scenario
    # The function must be called after mobility model is defined

   $ns_ initial_node_pos $node_($i) 35
}



#Tell nodes when the simulation ends
for {set i 0} {$i < $val(nn) } {incr i} {
    $ns_ at $val(stop).0 "$node_($i) reset";
}

$ns_ at  $val(stop).0002 "puts \"NS EXITING...\" ; $ns_ halt"

puts $tracefd "Lennart Wrote this!"
puts $tracefd "M 0.0 nn $val(nn) x $val(x) y $val(y) rp $val(rp)"
puts $tracefd "M 0.0 sc $val(sc) cp $val(cp) seed $val(seed)"
puts $tracefd "M 0.0 prop $val(prop) ant $val(ant)"

puts "Starting Simulation..."

$ns_ run
```

P.4 TrustManager (.h and .cc)

```
#ifndef TRUSTMANAGER_H
#define TRUSTMANAGER_H
#include <fstream.h>
#include <string>
#include <map>
#include "TrustValue.h"
#include "TrustUpdater.h"
#include "TrustFormater.h"
#include "TrustConstants.h"
//NS2 includes
#include "path.h"

/*
**      By Lennart Conrad, 12-10-2003
**  This class is used to store trustinformation about all known nodes
*/
class TrustFormater;

class TrustManager
{
 public:
  //constructor
  TrustManager();
  TrustManager(nsaddr_t id);//create a TM for the node with id
  //methods
  map<nsaddr_t,TrustValue*>& getTrustValues(void);
  TrustValue* getTrustValue(const nsaddr_t nid);
  void addTrustValue(nsaddr_t, TrustValue*);
  void createTrustValue(nsaddr_t, double);
  nsaddr_t getId();
  void setId(nsaddr_t n);//setter for id for this manager
  bool isKnown(nsaddr_t);
  void initNewTrustValues(Path& replyroute,nsaddr_t src);
  void initNewTrustValues(nsaddr_t replyroute[],int route_len, nsaddr_t src);
  double getTrustValueForNode(const nsaddr_t nid);

  //formate trust
  double formate();
  //update trust
  //double update(TrustValue*);
  void updateTrust(nsaddr_t nid, double event);//for one
  //for all known in route
  void updateTrustForKnownNodes(list<nsaddr_t>&,double event);
  void updateTrustForNodes(Path& path, double event);
  //TrustFormation
  TrustFormater* getTrustFormater();
  void setTrustFormater(TrustFormater*);
  //TrustUpdater
  TrustUpdater* getTrustUpdater();
  void setTrustUpdater(TrustUpdater*);
  //IO
  void toPrint(void);
  bool writeToFile(void);
  bool readFromFile(void);
  void closeRFile(void);//close file after read
  void closeWFile(void);//close file after write
  bool openRFile(int id);//for reading on start up
  bool openWFile(int id);//for writing
  //(de) serializing
  bool load(int id);//for loading
  bool save(int id);//for saving
  //testing
  void testIO();//writes some testinfo to file
  void testIt();
 private:
  map<nsaddr_t,TrustValue*> tvalues;//holds trust info for all known nodes
  nsaddr_t name;//id for this node
  ofstream outfile;//for writing
  ifstream infile; //for reading
  TrustUpdater* trustupdater;//the calcultare for updating trust
  TrustFormater* trustformater;//used when nodes are first encountered
  FILE* fdata;
};
//Do Not forget the trailing semi-colon
```

```
#endif //TRUSTMANAGER_H

#include "TrustManager.h"
#include <string>
#include <iostream>
#include <sstream>

TrustManager::TrustManager()
{}

TrustManager::TrustManager(nsaddr_t n)
{
  this->name = n;
}
//getter for name (id)
nsaddr_t TrustManager::getId()
{
  return this->name;
}

//setter for id
void TrustManager::setId(nsaddr_t n)
{
  this->name = n;
}


//updates trust for all known nodes in the path according to the eventtype
void TrustManager::updateTrustForKnownNodes(list<nsaddr_t>& known_route, double event)
{
  //just go through and call updatetrust for the nodes we know
  list<nsaddr_t>::iterator i;
  for(i=known_route.begin(); i != known_route.end(); ++i)
    {
      //cerr <<"updating trust TM"<<endl;
      this->updateTrust(*i, event);
    }
  //cerr <<"Finished updating trust TM"<<endl;
}


//updates trust for all nodes in the path according to the eventtype
void TrustManager::updateTrustForNodes(Path& path, double event)
{
  //just go through and call updatetrust for the nodes we know

  //NOTE ! it is important that i = 1 because evil nodes send
  // data packets as well
  for(int i=1; i <path.length() ; i++)
    {
      //cerr <<"updating trust TM"<<endl;
      int id = path[i].getNSAddr_t();
      //evil nodes id are in this array
      //for now they are hardcoded but they should be read from a file
      nsaddr_t evilnodes[] = {1,2,7,10,16,19,23,13,24,5};//max of 10
      //rmnsaddr_t evilnodes[] = {14,12,20,18,11,3,19,22,13,5};
      for(int j = 0; j<NROFEVILNODES;j++)
        {
          if(id == evilnodes[j] && i <path.length()-1)
            {
              //cerr<< "We are updating trust for a evil node:"<<id <<endl;
            }
        }

      this->updateTrust(id, event);

    }
  if(TRUSTVALUEUPDATEVERBOSE)//testing
    cout <<"Node: "<<this->name<<"Finished updating for route: "<<path.dump()<<endl;
}



//updates trust for the node with id according to the eventtype
void TrustManager::updateTrust(nsaddr_t nid, double event)
{
  //get the new value from the trustupdater
  if(isKnown(nid))
    {
      if(TRUSTVALUEUPDATEVERBOSE)//testing
       cerr << "Node" << this->name  <<"is updating trust
       for......................:"<<nid <<endl;
      //this->update(getTrustValue(nid), event);
      if(nid == this->name)//Always assign max trust to ourselves
        {
          double res = this->trustupdater->update(getTrustValue(nid), 1.0);
        }
      else
        {
          double res = this->trustupdater->update(getTrustValue(nid), event);
        }
      //cerr << "REs "<<res <<endl;

    }
  else
    {
      //cerr << "a node was not known in TrustManager::updateTrust" << endl;
      //establis trust with some standard value
      //cerr << "Node" << this->name  <<"is establishing trust for node:"<<nid <<endl;
      cerr << "valie of: S1_DEFAULTTRUSTVALUE" <<S1_DEFAULTTRUSTVALUE<<endl;
      this->createTrustValue(nid,S1_DEFAULTTRUSTVALUE);
    }
}


//Trustformater getter
TrustFormater* TrustManager::getTrustFormater()
{
  return this->trustformater;
}

//TrustFormater setter
void TrustManager::setTrustFormater(TrustFormater* tm)
{
       this->trustformater = tm;
}

//TrustUpdater getter
TrustUpdater* TrustManager::getTrustUpdater()
{
  return this->trustupdater;
}

//TrustUpdater setter
void TrustManager::setTrustUpdater(TrustUpdater* tu)
{
  this->trustupdater = tu;
}

//returns all trustvalues
map<nsaddr_t,TrustValue*>& TrustManager::getTrustValues()
{
  return tvalues;
}


//return the value from trustvalue for a node with id = nid
double TrustManager::getTrustValueForNode(const nsaddr_t nid)
{
       //LC This should not happen but i does!
  map<nsaddr_t,TrustValue*>::iterator it;
  if(!isKnown(nid) ) //if(it == tvalues.end())
       {
         cerr <<"Error gettrustValueForNode()"<<endl;
         cerr << "Node: "<<this->name << " does not have any trust value for
"<<nid<<endl;
         return 0.0;
       }
  else
    {
      return tvalues[nid]->getValue();
    }
  //return ( (double)rand()/(double)RAND_MAX+1 );
```

```
}

//return the trustvalue for a node with id = nid
TrustValue* TrustManager::getTrustValue(const nsaddr_t nid)
{
  //LC This should not happen but i does!
  //map<nsaddr_t,TrustValue*>::iterator it;

  if(!isKnown(nid) ) //it == tvalues.end())
    {
      //cerr <<"ERROR  getTrustValue "<<endl;
      //cerr << "Node: "<<this->name << " does not have any trust value for
"<<nid<<endl;
      //this happens when node X gets a route with node X (itself)therefore we assign
a high trustvalue
      tvalues[nid] = new TrustValue(nid,0.9);  //we have to create it!
      //cerr << "But now it has the value " << tvalues[nid]->getValue()<<endl;
    }
  return tvalues[nid];
}

//adds a the trutsvalue tval wwith the key id- WE NEED IT FOR LOADING!
void TrustManager::addTrustValue(nsaddr_t id, TrustValue* tval)
{
  if(!isKnown(id))
    {
      tvalues[id] = tval;
    }
  //LC : note could be a possible memory leak since the arg TrustValue* is not deleted
  //if the value allready exist
}
//creates and adds a new TrustValue to this trustmanager
void TrustManager::createTrustValue(nsaddr_t id, double tval)
{
  if(!isKnown(id))
    {
      tvalues[id] = new TrustValue(id,tval);
    }
}

//prints the cotent of the Trustmanager
void TrustManager::toPrint()
{
  map<nsaddr_t,TrustValue*>::iterator it;
  cout << "The TrustManager contains:\n" << endl;
  for(it = tvalues.begin();it!= tvalues.end();it++)
    {
            (it->second)->toPrint();
    }
}


//Saves all data so it can be loaded again later
bool TrustManager::save(nsaddr_t id)
{
  ostringstream ss;
  //create file name
  ss << id << ".dtmf";
  cout << "and the number was" << ss.str() << endl;
  //open the file, if it allredy exists it is deleted
  cout << "Opening file: " << ss.str() <<" for writing.............." << endl;
  fdata = fopen(ss.str().c_str(),"w");
  //did it open ok
  int nroftrustdata;
  if(fdata == NULL)//error on opening file
    {
      cout << "Error on opening file: " << ss.str() <<".dtmf in save()" << endl;
      return false;
    }
  else//ok
    {
      cout << "File: " << ss.str() <<" opened for writing by save" << endl;
      nroftrustdata = tvalues.size();//nr of stored trustvalues
      fwrite(&nroftrustdata,sizeof(nroftrustdata),1,fdata);
      //go through the map and let tvalues store them selves
      map<nsaddr_t,TrustValue*>::iterator it;
      //go through the manager an let the TrustValues handle IO
      for(it = tvalues.begin();it!=tvalues.end();it++)
        {
          (it->second)->save(fdata);
        }

    }
  fclose(fdata);
  cout << "Saved ok with"<< nroftrustdata <<"nodes saved" << endl;
  return true;
}

//load trustvalues
bool TrustManager::load(nsaddr_t id)
{
  ostringstream ss;
  //create file name
  ss << id << ".dtmf";
  cout << "and the number was" << ss.str() << endl;
  //open the file, if it allredy exists it is deleted
  cout << "Opening file: " << ss.str() <<" for loading.............." << endl;
  fdata = fopen(ss.str().c_str(),"r");
  //did it open ok
  if(fdata == NULL)//error on opening file
    {
      cout << "Error on opening file: " << ss.str() <<".dtmf in TrustManager::load()"
<< endl;
      return false;
    }
  else//ok
       {
         cout << "File: " << ss.str() <<" opened for loading by TrustManager::load()"
<< endl;
         int nroftrustdata;
         fread(&nroftrustdata,sizeof(int),1,fdata);//nr of stored trustvalues
         cout << "Load contains: " << nroftrustdata  <<" Trustvalues,
TrustManager::load()" << endl;
         //go through the map and let tvalues store them selves
         //go through the manager an let the TrustValues handle IO
         for(int i =0;i < nroftrustdata;i++)
           {
             TrustValue* v = new TrustValue();
             //let it load it self
             v->load(fdata);
             //store it
             addTrustValue(v->getId(), v);
               }
        }
  fclose(fdata);

  //test by printing content
  //this->toPrint();

        return true;
}


//open a . tmf file for the node to write trust info to
bool TrustManager::openWFile(nsaddr_t id)
{
  ostringstream ss;
  //create file name
  ss << id << ".tmf";
  cout << "and the number was" << ss.str() << endl;
  //open the file, if it allredy exists it is deleted
  cout << "Opening file: " << ss.str() <<" for writing.............." << endl;
  outfile.open(ss.str().c_str(),ios::out | ios::app | ios::trunc);
  //did it open ok
  if(!outfile.is_open())
    {
      cout << "Error on opening file: " << ss.str() <<".tmf" << endl;
               return false;
    }
  cout << "File: " << ss.str() <<" opened for writing" << endl;

  return true;
}
```

```
//open a . tmf file for the node to Read trust info from
bool TrustManager::openRFile(nsaddr_t id)
{
  ostringstream ss;
  //create file name
  ss << id << ".tmf";
  //open the file
  ifstream infile;
  //char buf[100];
  infile.open(ss.str().c_str());
  //read everything
  if(!infile.is_open())
    {
      cout << "Error: file:" << id << ".tmf could not be opened" << endl;
      return false;
    }
  cout << "Infile" << id << ".tmf opened ok!" << endl;
  return true;
}

//closes the in file
void TrustManager::closeRFile()
{
  infile.close();
}

//closes the outfile
void TrustManager::closeWFile()
{
  outfile.close();
}
//Writes the content of the trust manager to a file
bool TrustManager::writeToFile()
{
  //we better check that it's opened
  if(!outfile.is_open())
    {
       cerr << "Out file is not opened" << endl;
       return false;
    }
  else
    {
      map<nsaddr_t,TrustValue*>::iterator it;
      outfile << "Trustmanager for Node:" << name <<"\n";
      //go through the manager an let the TrustValues handle IO
      for(it = tvalues.begin();it!=tvalues.end();it++)
        {
          (it->second)->writeToFile(outfile);
        }
               //the file is NOT closed - remember to call closeWFile() !
    }
  return true;
}

bool TrustManager::readFromFile()
{
  //better tjeck that it is open
  char buf[100];
  if(infile.is_open())
        {
          cout << "File not  opened" << endl;
        }
  else
    {
      //read everything
      while(!infile.eof())
        {
        infile.read(buf,10);
        cout << "Read from infile"<< buf <<endl;
        }

    }
  return true;
}

//initialises trust values for unkown nodes by examining the entire path
//call with Path
```

```
void TrustManager::initNewTrustValues(Path& replyroute,nsaddr_t src)
{
  //cout << "TrustManager::initNewTrustValues.........." <<endl;
  this->trustformater->initNewTrustValues(replyroute,src);
  //cout << "TrustManager::initNewTrustValues Survided" <<endl;
}

//initialises trust values for unkown nodes by examining the entire path
// call with array
void TrustManager::initNewTrustValues(nsaddr_t replyroute[],int route_len, nsaddr_t
src)
{
        //cout << "TrustManager::initNewTrustValues.........." <<endl;
  this->trustformater->initNewTrustValues(replyroute,route_len,src);
  //cout << "TrustManager::initNewTrustValues Survided" <<endl;
}



//does this entry exist -> the node is known
bool TrustManager::isKnown(nsaddr_t id)
{
  map<nsaddr_t,TrustValue*>::iterator it;
  it = tvalues.find(id);//if id doesn't exist it returns iterator to the end
  return(!(it == tvalues.end()));
}

//for internal testing
void TrustManager::testIt()
{
  cout << "Trustmanager testIt() called" << endl;
}


//writes some testinfo to file
void TrustManager::testIO()
{
  //we better check that it's opened
  if(!outfile.is_open())
    {
        cerr << "Out file is not opened (testIO())" << endl;
    }
  else
    {

      cout << "Trustmanager for Node:" << name <<"created!\n" << endl;
      outfile << "Trustmanager for Node:" << name <<"created!\n";
      outfile << "Trustmanager for Node:" << name <<"created!\n";
      outfile << "Trustmanager for Node:" << name <<"created!\n";
      closeWFile();
    }
}
```

P.5 TrustFormater (.h and .cc)

```
#ifndef TRUSTFORMATER_H
#define TRUSTFORMATER_H
#include "TrustValue.h"
#include "TrustManager.h"
#include "TrustConstants.h"

/*
**      By Lennart Conrad 12-10-2003
**  The TrustFormater class is an abstract class.
**      implementation are found in TrustFormater.cpp
**
*/

class TrustManager;// we need this declaration here
class TrustFormater
{
 public:
  //Constructor
  //TrustFormater();
  //methods
  //first argument is the trustValue that needs updating and holds data, secodn is the
experience value
  //virtual double update(TrustValue&, double)=0;//abstract methods
  virtual void initNewTrustValues(const Path& replyroute,const nsaddr_t src) = 0;
  virtual void initNewTrustValues(const nsaddr_t replyroute[],const int route_len,
const nsaddr_t src) = 0 ;

  //fields
  TrustManager* trustmanager;
};

//implementing class - strategy 1
class TrustFormaterS1 : public TrustFormater
{
 public:
        //Constructor
  TrustFormaterS1();
  TrustFormaterS1(TrustManager*);
  //methods
  void initNewTrustValues(const Path& replyroute,const nsaddr_t src);
  void initNewTrustValues(const nsaddr_t replyroute[],const int route_len, const
nsaddr_t src);
  //fields
};

//implementing class - strategy 2
class TrustFormaterS2 : public TrustFormater
{
 public:
  //Constructor
  TrustFormaterS2();
  TrustFormaterS2(TrustManager*);
  //methods
  void initNewTrustValues(const Path& replyroute,const nsaddr_t src);
  void initNewTrustValues(const nsaddr_t replyroute[],const int route_len, const
nsaddr_t src);

  //fields
};

#endif //TRUSTFORMATER_H

#include "TrustFormater.h"
#include "TrustValue.h"
#include "TrustConstants.h"

//---------- Strategy 1 -----------------------
TrustFormaterS1::TrustFormaterS1()
{}

//we need to be able to reference trustmanager to getinfo on known nodes
TrustFormaterS1::TrustFormaterS1(TrustManager* tm)
{
  this->trustmanager = tm;
}

//initializes trust for unknown nodes from known nodes (Call with nsaddr_t[])
void TrustFormaterS1::initNewTrustValues(const nsaddr_t replyroute[],const int
route_len, const nsaddr_t src)
{}

//initializes trust for unknown nodes from known nodes (Call with Path)
void TrustFormaterS1::initNewTrustValues(const Path& replyroute, const nsaddr_t src)
{

  //cout << "TrustFormater Called" << endl;
  //This is how is done:
  //1 go through the nodes and indentify the known ones ->
  //put the unknown in one list and the known in anoter
  //2 find min value of known nodes trust values
  //3 create new Trustvalues with the minimum value
  list<nsaddr_t> known;
  list<nsaddr_t> unknown;
  //cout << "DId 2 list" << endl;
  //1 find known/unknown
  for(int i =0;i < replyroute.length();i++)//1
    {
      //cout << "loop 1" << endl;
      nsaddr_t id = replyroute[i].getNSAddr_t();
      //cout << "loop 2" << endl;
      if(trustmanager->isKnown(id))//we know it
        {
          //cout << "loop 3" << endl;
          known.push_front(id);
        }
      else//we don't know it
        {
        unknown.push_front(id);
        //cout << "loop 4" << endl;
        }
    }
        //cout << "FINISHED DOING TrustFormation Part 1" << endl;

        //2. find minimum tval of know nodes
  list<nsaddr_t>::iterator it;
  double ghost = 1000.0;//A very high value
        double min = ghost;

        for(it = known.begin(); it != known.end(); it++)
          {
            int nid = *it;
            if(trustmanager->getTrustValue(nid)->getValue() < min)
                {
                  min = trustmanager->getTrustValue(nid)->getValue();
                }
          }
        //cout << "FINISHED DOING TrustFormation Part 2" << endl;
        //tjeck if min was ever set
        if(min == ghost)
          {
            //cerr << "valie of: S1_DEFAULTTRUSTVALUE in trustformater"
<<S1_DEFAULTTRUSTVALUE<<endl;
                min = S1_DEFAULTTRUSTVALUE; //if we dont know anybody it should be set
to some standard val
          }

        //3 now create new Trustvalues
        //list<nsaddr_t>::iterator uit;

        for(it = unknown.begin(); it != unknown.end(); it++)
          {
            trustmanager->createTrustValue(*it, min);
          }
        //cout << "FINISHED DOING TrustFormation Part 3" << endl;
}


//------------ Strategy 2 ------------------------
TrustFormaterS2::TrustFormaterS2()
{}

TrustFormaterS2::TrustFormaterS2(TrustManager* tm)
{
        this->trustmanager = tm;
}

//initializes trust for unknown nodes from known nodes (Call with nsaddr_t[])
void TrustFormaterS2::initNewTrustValues(const nsaddr_t replyroute[],const int
route_len, const nsaddr_t src)
{}

//initializes trust for unknown nodes from known nodes (Call with Path)

/* strategy 2 - assign the average trust of the route*/
void TrustFormaterS2::initNewTrustValues(const Path& replyroute,const nsaddr_t src)
{
  //cout << "TrustFormater Called" << endl;
  //This is how is done:
  //1 go through the nodes and indentify the known ones ->
  //put the unknown in one list and the known in anoter
  //2 find min value of known nodes trust values
  //3 create new Trustvalues with the minimum value
  list<nsaddr_t> known;
  list<nsaddr_t> unknown;
  //cout << "DId 2 list" << endl;
  //1 find known/unknown
  for(int i =0;i < replyroute.length();i++)//1
    {
      //cout << "loop 1" << endl;
      nsaddr_t id = replyroute[i].getNSAddr_t();
      //cout << "loop 2" << endl;
      if(trustmanager->isKnown(id))//we know it
        {
          //cout << "loop 3" << endl;
          known.push_front(id);
        }
      else//we don't know it
        {
          unknown.push_front(id);
          //cout << "loop 4" << endl;
        }
    }
  //cout << "FINISHED DOING TrustFormation Part 1" << endl;

  //2. find minimum tval of know nodes
  list<nsaddr_t>::iterator it;
  double ghost = 1000.0;//A very high value
  double average = ghost;

        for(it = known.begin(); it != known.end(); it++)
          {
               int nid = *it;
               average =  average + trustmanager->getTrustValue(nid)->getValue();
          }
        //cout << "FINISHED DOING TrustFormation Part 2" << endl;
        //tjeck if min was ever set
        if( average == ghost)
          {
        //cerr << "valie of: S2_DEFAULTTRUSTVALUE in trustformater"
<<S2_DEFAULTTRUSTVALUE<<endl;
            average = S2_DEFAULTTRUSTVALUE; //if we dont know anybody it should be set
to some standard val
        }
        else
        {
          average = average/known.size();
        }
        //3 now create new Trustvalues
        //list<nsaddr_t>::iterator uit;

        for(it = unknown.begin(); it != unknown.end(); it++)
          {
            trustmanager->createTrustValue(*it, average);
          }
        //cout << "FINISHED DOING TrustFormation Part 3" << endl;

}
```

P.6 TrustUpdater (.h and .cc)

```
#ifndef TRUSTUPDATER_H
#define TRUSTUPDATER_H
#include "TrustValue.h"
/*
**      By Lennart Conrad 12-10-2003
**  The TrustUpdater class is an abstract class.
**      Subclasses implementing different strategies should impmement the calculate()
methods
**      implementation are found in TrustUpdater.cpp
**
*/
class TrustUpdater
{
 public:
  //Constructor
  //TrustUpdater();
  //methods
  //first argument is the trustValue that needs updating and holds data, secodn is the
experience value
  virtual double update(TrustValue*, double)=0;//abstract methods
};
//implementing class - strategy 1
class TrustUpdaterS1 : public TrustUpdater
{
 public:
  //Constructor
  TrustUpdaterS1();
  //methods
  double update(TrustValue*,double);
};
//implementing class - strategy 2
class TrustUpdaterS2 : public TrustUpdater
{
 public:
  //Constructor
  TrustUpdaterS2();
  //methods
  double update(TrustValue*,double);

};
#endif //TRUSTUPDATER_H

#include "TrustUpdater.h"
#include "TrustValue.h"

TrustUpdaterS1::TrustUpdaterS1()
{}

TrustUpdaterS2::TrustUpdaterS2()
{}

double TrustUpdaterS1::update(TrustValue* tval, double exp)
{
  //parameters
  double dpos = 0.9; //positive inflation
  double dneg = 0.9; //negative inflation
  double ret;
  //update trustvalue + nr of experiences
  if(exp ==    ACKRECEIVED )//positive experineces
    {
      ret = (dpos*tval->getValue() + (1.0 - dpos)*exp);
      //assign the new value
      tval->setValue(ret);
      tval->addExperience(exp);
      //cerr << "Trust increased: " << ret << endl;
    }
  else if(exp == DATAPACKETRECEIVED)//
    {
      ret = (dpos*tval->getValue() + (1.0 - dpos)*exp);
      //assign the new value
      tval->setValue(ret);
      tval->addExperience(exp);
      //cerr << "Trust increased: " << ret << endl;

      /*LC debug/analysis -  to see if we are giving trust to evil nodes*/

      //evil nodes id are in this array
      //for now they are hardcoded but they should be read from a file
      /*nsaddr_t evilnodes[] = {1,2,7,10,16,19,23,13,24,5};//max of 10
      //rmnsaddr_t evilnodes[] = {14,12,20,18,11,3,19,22,13,5};
      for(int i = 0; i<NROFEVILNODES;i++)
      {
      if(tval->getId() == evilnodes[i])
      {
      cerr<< "We are updating trust for a evil node!" <<endl;
      }
      } */
    }
  else//NOACKRECEIVED
```

```
    {
      //cout << "tval->getValue():"<< tval->getValue() << endl;
      ret = (dneg*tval->getValue() + (1.0 - dneg)*exp);
      tval->setValue(ret);
      tval->addExperience(exp);
      //cerr << "Trust decreased: " << ret << endl;
    }
  return ret;//if somebody wants to see the updated value
}

double TrustUpdaterS2::update(TrustValue* tval, double exp)
{

  //parameters
  double dpos = 0.9; //positive inflation
  double dneg = 0.9; //negative inflation
  double ret;
  double retexp;

  //update trustvalue + nr of experiences
  if(exp ==    ACKRECEIVED )//positive experineces
    {
      ret = (dpos*tval->getValue() + (1.0 - dpos)*exp);
      retexp = tval->getAverageOfExperiences();
      ret = (ret + retexp)/2.0;
      //assign the new value
      tval->setValue(ret);
      tval->addExperience(exp);
      //cerr << "Trust increased: " << ret << endl;
        }
  else if(exp == DATAPACKETRECEIVED)//
    {
      ret = (dpos*tval->getValue() + (1.0 - dpos)*exp);
      retexp = tval->getAverageOfExperiences();
      ret = (ret + retexp)/2.0;
      //assign the new value
      tval->setValue(ret);
      tval->addExperience(exp);
      //cerr << "Trust increased: " << ret << endl;
    }
  else//NOACKRECEIVED
    {
      //cout << "tval->getValue():"<< tval->getValue() << endl;
      ret = (dneg*tval->getValue() + (1.0 - dneg)*exp);
      retexp = tval->getAverageOfExperiences();
      ret = (ret + retexp)/2.0;
      tval->setValue(ret);
      tval->addExperience(exp);
      //cerr << "Trust decreased: " << ret << endl;

    }
  return ret;//if somebody wants to see the updated value
}
```

P.7 ACKMonitor (.h and .cc)

```
#ifndef ACKMONITOR_H
#define ACKMONITOR_H
#include <map>
#include <list>
#include "TrustManager.h"
//NS2 includes
#include "path.h"
#include <scheduler.h>
#include "TrustConstants.h"
using namespace std;
/*
**      Created by Lennart Conrad 23-10-2003
**      These classes are used to handle acknowledgments
**      in the trust extensions to DSR
**
*/

//class used to store data about a send ACK
class ACKData
{
 public:
  ACKData();
  ACKData(double ackid,nsaddr_t route[],int routel, Time send_at);
  //fields
  double ackid;
  list<nsaddr_t> route;//all ids on the route
  Time sendat;//time we send this ACK
  //methods
  inline double getACKId() { return ackid; }
  inline Time getTime() { return sendat; }
  inline int getLength() {return route.size(); }

};


class ACKMonitor
{
 public:
  //Constructor
  ACKMonitor();
  ACKMonitor(TrustManager*);

  //Destructor
  ~ACKMonitor();
  ACKData* getACK(double ackid);
  void setTrustManager(TrustManager*);
  void addACK(double ackid,nsaddr_t route[],int routelen, Time send_at);
  //Returns True if we have the ACK reqisterred
  bool isACKRegistered(double ackid);
  void removeACK(double id);
  void handleACKReceived(double ackid,nsaddr_t returnroute[],int rl, Time rec_at,
nsaddr_t from);
  void scanForOldACKs();
  //for testing
  void terminate();
  void testIt();
  //fields
 private:
  map<double,ACKData*> acks;//holds all ACKData
  TrustManager* trust_manager;
  //testing

  //these are just for statistics on how many acks are recieved/send etc
  int ack_time_out;
  int ack_rec_ok;
  int ack_add;
  int dup_add_ack;
  int ack_to_late;
};
   //Do Not forget the trailing semi-colon
#endif //ACKMONITOR_H

#ifndef ACKMONITOR_H
#define ACKMONITOR_H
#include <map>
#include <list>
#include "TrustManager.h"
//NS2 includes
#include "path.h"
#include <scheduler.h>
#include "TrustConstants.h"
using namespace std;
/*
**      Created by Lennart Conrad 23-10-2003
**      These classes are used to handle acknowledgments
**      in the trust extensions to DSR
**
*/

//class used to store data about a send ACK
class ACKData
{
 public:
  ACKData();
  ACKData(double ackid,nsaddr_t route[],int routel, Time send_at);
  //fields
  double ackid;
  list<nsaddr_t> route;//all ids on the route
  Time sendat;//time we send this ACK
  //methods
  inline double getACKId() { return ackid; }
  inline Time getTime() { return sendat; }
  inline int getLength() {return route.size(); }

};


class ACKMonitor
{
 public:
  //Constructor
  ACKMonitor();
  ACKMonitor(TrustManager*);

  //Destructor
  ~ACKMonitor();
  ACKData* getACK(double ackid);
  void setTrustManager(TrustManager*);
  void addACK(double ackid,nsaddr_t route[],int routelen, Time send_at);
  //Returns True if we have the ACK reqisterred
  bool isACKRegistered(double ackid);
  void removeACK(double id);
  void handleACKReceived(double ackid,nsaddr_t returnroute[],int rl, Time rec_at,
nsaddr_t from);
  void scanForOldACKs();
  //for testing
  void terminate();
  void testIt();
  //fields
 private:
  map<double,ACKData*> acks;//holds all ACKData
  TrustManager* trust_manager;
  //testing

  //these are just for statistics on how many acks are recieved/send etc
  int ack_time_out;
  int ack_rec_ok;
  int ack_add;
  int dup_add_ack;
  int ack_to_late;
};
   //Do Not forget the trailing semi-colon
#endif //ACKMONITOR_H
```

P.8 RouteSelector (.h and .cc)

```
#ifndef ROUTESELECTOR_H
#define ROUTESELECTOR_H
#include "TrustManager.h"
//NS2 includes
#include "path.h"
//#include <scheduler.h>

using namespace std;
/*
**      Created by Lennart Conrad 30-10-2003
**      These classes are used to handle route selection
**      in the trust extensions to DSR
**
*/
class RouteSelector
{

 public:
  TrustManager* trust_manager;
  //methods
  double virtual evaluateRoute(Path&,int&) = 0;
  void virtual setTrustManager(TrustManager* tm)=0;
  void virtual testIt()=0;
};

class RouteSelectorS1 : public RouteSelector
{
 public:
  //Constructor
  RouteSelectorS1();
  RouteSelectorS1(TrustManager*);
  //Destructor
  ~RouteSelectorS1();
  //methods
  double evaluateRoute(Path&,int&);
  void setTrustManager(TrustManager* tm);
  void testIt();
};

class RouteSelectorS2 : public RouteSelector
{
 public:
  //Constructor
  RouteSelectorS2();
  RouteSelectorS2(TrustManager*);
  //Destructor
  ~RouteSelectorS2();
  //methods
  double evaluateRoute(Path&,int&);
  void setTrustManager(TrustManager* tm);
  void testIt();
};

class RouteSelectorS3 : public RouteSelector
{
 public:
  //Constructor
  RouteSelectorS3();
  RouteSelectorS3(TrustManager*);
  //Destructor
  ~RouteSelectorS3();
  //methods
  double evaluateRoute(Path&,int&);
  void setTrustManager(TrustManager* tm);
  void testIt();
};

class RouteSelectorS4 : public RouteSelector
{
 public:
  //Constructor
  RouteSelectorS4();
  RouteSelectorS4(TrustManager*);
  //Destructor
  ~RouteSelectorS4();
  //methods
  double evaluateRoute(Path&,int&);
  void setTrustManager(TrustManager* tm);
  void testIt();
};

class RouteSelectorS5 : public RouteSelector
{
 public:
  //Constructor
  RouteSelectorS5();
  RouteSelectorS5(TrustManager*);
  //Destructor
  ~RouteSelectorS5();
  //methods
  double evaluateRoute(Path&,int&);
  void setTrustManager(TrustManager* tm);
  void testIt();
};

 //Do Not forget the trailing semi-colon
#endif //ROUTESELECTOR_H

#include <iostream.h>
```

```
#include <stdio.h>
#include <string>
#include <map>
#include <sstream>//for double to string
#include "RouteSelector.h"
#include "TrustConstants.h"

using namespace std;

ofstream routefile("routestat.txt");
//------------ Route selection strategy 1 -----------------/
//Constructor
RouteSelectorS1::RouteSelectorS1(){}

RouteSelectorS1::RouteSelectorS1(TrustManager* tm)
{
  this->trust_manager = tm;
}

//Destructor
RouteSelectorS1::~RouteSelectorS1(){}

//setter for Trustmanager
void RouteSelectorS1::setTrustManager(TrustManager* tm)
{
  this->trust_manager = tm;
}

//returns the calculated trust of the route
//Strategy 1: average of all trust/nr of nodes in path
double RouteSelectorS1::evaluateRoute(Path& route,int& stopat)
{
  //routefile << "Node :"<<trust_manager->getId() <<"wrote to file\n";
  double result = 0.0;
  double total_trust =0.0;
  double average_trust = 0.0;
  int total_nr_of_nodes=0;
  double temp =0.0;
  //the route might be A->B->C->D->E even though e only want to use
  //A->B->C, the stopat arg tells us were the destination is-so we shorten the route
to this point
  //route.removeSection(stopat,route.length());
  route.setLength(stopat+1);

  //the sender is always the first in the ID[] but we take it with anyway

  for(int i =0; i < stopat;i++)
    {

      if(stopat == 1)//dest is next to us
        {
          //cerr << "destination is next hop"<<endl;
          return MAXTRUSTVAL;
        }

      //LC this should NEVER happen
      if(route.length() < stopat)
        {
          cerr<<"\n The biggest error in the hole world ever has ocurred\n "<<endl;
          cout<<"\n The biggest error in the hole world ever has ocurred\n "<<endl;
          return -1.0;
        }


      if(trust_manager->isKnown(route[i].getNSAddr_t()))//we know it
        {
          //should maybe consult a policy before we add the value?
          //int h = route[i].getNSAddr_t();
          //cerr << "DOING IT\n" <<endl;
          //temp = ( (double)rand()/(double)RAND_MAX+1 );//LC debugtrust_manager-
>getTrustValue(route[i].getNSAddr_t())->getValue();
          //temp = trust_manager->getTrustValueForNode(route[i].getNSAddr_t());
          temp = trust_manager->getTrustValue(route[i].getNSAddr_t())->getValue();
          //cerr << "The value of the temp variable is :" << temp << endl;
          total_trust = total_trust + temp;
          //cerr << "DID IT\n" <<endl;
          total_nr_of_nodes++;
        }
      else//should not happen but if it does we create it
        {
        cerr << "Node: "<< route[i].getNSAddr_t() <<" unknown in path in
RouteSelectorS1::evaluateRoute" <<endl;
        trust_manager->createTrustValue(route[i].getNSAddr_t(),DEFAULTUNKNOWNVAL);
        temp = trust_manager->getTrustValue(route[i].getNSAddr_t())->getValue();
        total_trust = total_trust + temp;
                       total_nr_of_nodes++;
        }

    }
        //now do the heuristic
  average_trust = total_trust/total_nr_of_nodes;
        result = average_trust;//total_nr_of_nodes;//maybe divide here
        if(RSVERBOSE)
        cerr << "route selector S1 calculated a route to the value: "<<result<<"and the
lenght of the route was:"<<stopat<<endl;
        return result;
}

//used for testing
void  RouteSelectorS1::testIt()
{

  cout << "Testing RouteSelector S1" << endl;
  cout << "Testing Call to Trust Manager testIO()" << endl;
  trust_manager->testIt();
}

//------------ Route selection strategy 2 -----------------/

//Constructor
RouteSelectorS2::RouteSelectorS2(){}

RouteSelectorS2::RouteSelectorS2(TrustManager* tm)
{
  this->trust_manager = tm;
}

//Destructor
RouteSelectorS2::~RouteSelectorS2(){}

//setter for Trustmanager
void RouteSelectorS2::setTrustManager(TrustManager* tm)
{
  this->trust_manager = tm;
}

//Strategy 2: average trust/nr_of_nodes -> shorter routes are favoured
double RouteSelectorS2::evaluateRoute(Path& route,int& stopat)
{
  double result = 0.0;
  double total_trust =0.0;
  double average_trust = 0.0;
  int total_nr_of_nodes=0;
  double temp =0.0;
  //the route might be A->B->C->D->E even though e only want to use
  //A->B->C, the stopat arg tells us were the destination is-so we shorten the route
to this point
  //route.removeSection(stopat,route.length());
  route.setLength(stopat+1);
  //the sender is always the first in the ID[] but we take it with anyway

  for(int i =0; i < stopat;i++)
    {

      if(stopat == 1)//dest is next to us
        {
          //cerr << "destination is next hop"<<endl;
          return MAXTRUSTVAL;
        }

      //LC this should NEVER happen
      if(route.length() < stopat)
        {
          cerr<<"\n The biggest error in the hole world ever has ocurred\n "<<endl;
          cout<<"\n The biggest error in the hole world ever has ocurred\n "<<endl;
          return -1.0;
        }


      if(trust_manager->isKnown(route[i].getNSAddr_t()))//we know it
               {
                 //should maybe consult a policy before we add the value?
                      //int h = route[i].getNSAddr_t();
                      //cerr << "DOING IT\n" <<endl;
                      //temp = ( (double)rand()/(double)RAND_MAX+1 );//LC
debugtrust_manager->getTrustValue(route[i].getNSAddr_t())->getValue();
                      //temp = trust_manager-
>getTrustValueForNode(route[i].getNSAddr_t());
                 temp = trust_manager->getTrustValue(route[i].getNSAddr_t())-
>getValue();
                 //cerr << "The value of the temp variable is :" << temp << endl;
                 total_trust = total_trust + temp;
                 //cerr << "DID IT\n" <<endl;
                 total_nr_of_nodes++;
               }
      else//should not happen but if it does we create it
        {
          cerr << "Node: "<< route[i].getNSAddr_t() <<" unknown in path in
RouteSelectorS2::evaluateRoute" <<endl;
          trust_manager->createTrustValue(route[i].getNSAddr_t(),DEFAULTUNKNOWNVAL);
          temp = trust_manager->getTrustValue(route[i].getNSAddr_t())->getValue();
                      total_trust = total_trust + temp;
                      total_nr_of_nodes++;
        }

        }
  //now do the heuristic
  average_trust = total_trust/total_nr_of_nodes;
  result = average_trust/total_nr_of_nodes;//maybe divide here
  if(RSVERBOSE)
    cerr << "route selector S2 calculated a route to the value: "<<result<<"and the
lenght of the route was:"<<stopat<<endl;
        return result;
}

//used for testing
void  RouteSelectorS2::testIt()
{

  cout << "Testing RouteSelector S2" << endl;
        cout << "Testing Call to Trust Manager testIO()" << endl;
        trust_manager->testIt();

}

//------------ Route selection strategy 3 -----------------/

//Constructor  using the average of the experiences and Not trust values
RouteSelectorS3::RouteSelectorS3(){}

RouteSelectorS3::RouteSelectorS3(TrustManager* tm)
{
  this->trust_manager = tm;
}

//Destructor
RouteSelectorS3::~RouteSelectorS3(){}

//setter for Trustmanager
void RouteSelectorS3::setTrustManager(TrustManager* tm)
{
  this->trust_manager = tm;
}

//Strategy 3: buidl on the number of experiences
double RouteSelectorS3::evaluateRoute(Path& route,int& stopat)
{
  double result = 0.0;
  double total_trust =0.0;
  double average_trust = 0.0;
  int total_nr_of_nodes=0;
  double temp= 0.0;
  //the route might be A->B->C->D->E even though e only want to use
  //A->B->C, the stopat arg tells us were the destination is-so we shorten the route
to this point
  //route.removeSection(stopat,route.length());
        route.setLength(stopat+1);

        //the sender is always the first in the ID[] but we take it with anyway

        for(int i =0; i < stopat+1;i++)
          {
        //LC this should NEVER happen
            if(route.length() < stopat)
              {
               cerr<<"\n The biggest error in the hole world ever has ocurred\n
"<<endl;
               cout<<"\n The biggest error in the hole world ever has ocurred\n
"<<endl;
               return -1.0;
              }
            if(stopat == 1)
              {
               //cerr << "destination is next hop"<<endl;
               return MAXTRUSTVAL;
              }

            if(trust_manager->isKnown(route[i].getNSAddr_t()))//we know it
              {
               //should maybe consult a policy before we add the value?

               temp = trust_manager->getTrustValue(route[i].getNSAddr_t())-
>getAverageOfExperiences();
               //cerr << "The value of the temp variable is :" << temp << endl;
               total_trust = total_trust + temp;
               //cerr << "DID IT\n" <<endl;
               total_nr_of_nodes++;
              }
               else//should not happen but if it does we create it
                 {
                   cerr << "Node: "<< route[i].getNSAddr_t() <<" unknown in path in
RouteSelectorS3::evaluateRoute" <<endl;
                   trust_manager-
>createTrustValue(route[i].getNSAddr_t(),DEFAULTUNKNOWNVAL);
                       temp = trust_manager->getTrustValue(route[i].getNSAddr_t())-
>getAverageOfExperiences();
                       total_trust = total_trust + temp;
                       total_nr_of_nodes++;
               }

          }
        //now do the heuristic

        average_trust = total_trust/total_nr_of_nodes;
        result = average_trust;
        if(RSVERBOSE)
        cerr << "route selector S3 calculated a route to the value: "<<result<<"and the
lenght of the route was:"<<stopat<<endl;
        return result;
}

//used for testing
void  RouteSelectorS3::testIt()
{

        cout << "Testing RouteSelector S3" << endl;
        cout << "Testing Call to Trust Manager testIO()" << endl;
        trust_manager->testIt();

}

//------------ Route selection strategy 4 -----------------/

//Constructor  using the average of the experiences and Not trust values
RouteSelectorS4::RouteSelectorS4(){}

RouteSelectorS4::RouteSelectorS4(TrustManager* tm)
{
        this->trust_manager = tm;
}

//Destructor
RouteSelectorS4::~RouteSelectorS4(){}

//setter for Trustmanager
void RouteSelectorS4::setTrustManager(TrustManager* tm)
{
 this->trust_manager = tm;
}

//Strategy 4: buidl on the number of experiences using a policy
double RouteSelectorS4::evaluateRoute(Path& route,int& stopat)
{
         double result = 0.0;
        double total_trust =0.0;
        double average_trust = 0.0;
        int total_nr_of_nodes=0;
        double temp=0.0;
        //the route might be A->B->C->D->E even though e only want to use
   //A->B->C, the stopat arg tells us were the destination is-so we shorten the route
to this point
        //route.removeSection(stopat,route.length());
        route.setLength(stopat+1);


        //the sender is always the first in the ID[] but we take it with anyway

        for(int i =0; i < stopat;i++)
        {
        if(stopat == 1)//dest is next to us
                {
                        //cerr << "destination is next hop"<<endl;
                        return MAXTRUSTVAL;
    }

    //LC this should NEVER happen

                if(route.length() < stopat)
                {
         cerr<<"\n The biggest error in the hole world ever has ocurred\n "<<endl;
                                cout<<"\n The biggest error in the hole world ever has
ocurred\n "<<endl;
                                return -1.0;
                }


                if(trust_manager->isKnown(route[i].getNSAddr_t()))//we know it
                {
                        //should maybe consult a policy before we add the value?

                        temp = trust_manager->getTrustValue(route[i].getNSAddr_t())-
>getAverageOfExperiences();
                        //cerr << "The value of the temp variable is :" << temp << endl;
                        //POLICY if trust < -0.25 return -1
                        if(temp < -0.25)
                        {
        return -1.0;
                        }
                        total_trust = total_trust + temp;
                        //cerr << "DID IT\n" <<endl;
                        total_nr_of_nodes++;
    }
                else//should not happen but if it does we create it
                {
        cerr << "Node: "<< route[i].getNSAddr_t() <<" unknown in path in
RouteSelectorS4::evaluateRoute" <<endl;
                        trust_manager-
>createTrustValue(route[i].getNSAddr_t(),DEFAULTUNKNOWNVAL);
                        temp = trust_manager->getTrustValue(route[i].getNSAddr_t())-
>getAverageOfExperiences();
                        //POLICY if trust < -0.25 return -1
                        if(temp < -0.25)
                        {
        return -1.0;
                        }
```

```
                      total_trust = total_trust + temp;
                      total_nr_of_nodes++;
              }

        }
        //now do the heuristic

        average_trust = total_trust/total_nr_of_nodes;
        result = average_trust;

        if(RSVERBOSE)
        cerr << "route selector S4 calculated a route to the value: "<<result<<"and the
lenght of the route was:"<<stopat<<endl;
        return result;
}

//used for testing
void  RouteSelectorS4::testIt()
{

        cout << "Testing RouteSelector S4" << endl;
        cout << "Testing Call to Trust Manager testIO()" << endl;
        trust_manager->testIt();

}

//------------ Route selection strategy 5 return minimum of trust values -------------
----/

//Constructor  using the average of the experiences and Not trust values
RouteSelectorS5::RouteSelectorS5(){}

RouteSelectorS5::RouteSelectorS5(TrustManager* tm)
{
        this->trust_manager = tm;
}

//Destructor
RouteSelectorS5::~RouteSelectorS5(){}

//setter for Trustmanager
void RouteSelectorS5::setTrustManager(TrustManager* tm)
{
 this->trust_manager = tm;
}

//Strategy 5: Return the lowest trust value of the route
double RouteSelectorS5::evaluateRoute(Path& route,int& stopat)
{
        double result = 0.0;
        double temp=0.0;
        int lowestid;
        double minimum;
        int firsttime = 1;
        //the route might be A->B->C->D->E even though e only want to use
   //A->B->C, the stopat arg tells us were the destination is-so we shorten the route
to this point
        //route.removeSection(stopat,route.length());
        route.setLength(stopat+1);


        //the sender is always the first in the ID[] but we take it with anyway
        //cerr<<"Node : "<< route[0].getNSAddr_t()<<"is evaluating"<< route.dump()
<<"with stopat: "<<stopat<<endl;
        for(int i =0; i < stopat;i++)
        {
        //LC this should NEVER happen
                /*if(route.length() < stopat)
                {
         cerr<<"\n The biggest error in the hole world ever has ocurred\n "<<endl;
                              cout<<"\n The biggest error in the hole world ever has
ocurred\n "<<endl;
                              return -1.0;
                }       */

    if(stopat == 1)
                {
                        //cerr << "destination is next hop"<<endl;
                        return MAXTRUSTVAL;
    }
                if(trust_manager->isKnown(route[i].getNSAddr_t()))//we know it
                {
                        //should maybe consult a policy before we add the value?
                        if(firsttime)
                        {
                                minimum = trust_manager-
>getTrustValue(route[i].getNSAddr_t())->getValue();
                                firsttime = 0;
                                lowestid = route[i].getNSAddr_t();
                        }
                        else
                        {
                                temp = trust_manager-
>getTrustValue(route[i].getNSAddr_t())->getValue();
                                if(temp < minimum)
                                {
                minimum = temp;
                                        lowestid = route[i].getNSAddr_t();
                                }
                        }
    }
                else//should not happen but if it does we create it
                {
        cerr << "Node: "<< route[i].getNSAddr_t() <<" unknown in path in
RouteSelectorS5::evaluateRoute" <<endl;
                        trust_manager-
>createTrustValue(route[i].getNSAddr_t(),DEFAULTUNKNOWNVAL);
                        if(firsttime)
                        {
                                minimum = trust_manager-
>getTrustValue(route[i].getNSAddr_t())->getValue();
                                firsttime = 0;
                        }
                        else
                        {
                                temp = trust_manager-
>getTrustValue(route[i].getNSAddr_t())->getValue();
                                if(temp < minimum)
                                {
                minimum = temp;
                                }
                        }
                }

        }
        if(RSVERBOSE) cerr << "RS 5 returning:"<<minimum<<"from node with
id"<<lowestid<<endl; ;
        return minimum;
}

//used for testing
void  RouteSelectorS5::testIt()
{

        cout << "Testing RouteSelector S4" << endl;
        cout << "Testing Call to Trust Manager testIO()" << endl;
        trust_manager->testIt();

}
```

P.9 TrustConstants.h

```
/*  -*- c++ -*-
    TrustConstants.h
    Constants used in trust, adjustment here makes it easier to change
    settings of simulation

*/

#ifndef TRUSTCONSTANTS_H
#define TRUSTCONSTANTS_H
```

```
#include "path.h"
//LC use these constants to define evil behaviuor
#define NROFEVILNODES 10                            //max 10

/*Acknowledgements */
#define AVR_ACK_TIME_OUT_VAL 0.07    //ACK time out in seconds - measured!
#define MAXWAIT 0.01                                             //what is the max
wait  we will add to the average

/*events -meaning packets received */

#define ACKRECEIVED 1.0                                    //ack received
#define NOACKRECEIVED -1.0                  //acktimed out
#define DATAPACKETRECEIVED 0.7       //data from someboby else
#define MAXTRUSTVAL 1.0 //the max trust value
/*used to give when no nodes in a route is known - trustformater*/
#define S1_DEFAULTTRUSTVALUE -0.4//the default trustval for s1 - estimated
#define S2_DEFAULTTRUSTVALUE -0.4 //for strategy 2
/*used when node is unknown is routeselector*/
#define DEFAULTUNKNOWNVAL -0.4       //the value we give to nodes that are unknown
during route selection (determined by experiments)

#define NOEXPERIENCES -0.4 //if we ask for the average of experiences and there are no
experinces (RS3)
#define MAX_NR_OF_EXP 5 //the number of experinces we store

#define USEROUTESELECTOR 4 //which routeselection to use

//verbose flags for viewing diffent output

#define RSVERBOSE 0 //1 debug routeselector
#define ACKVERBOSE 0
#define TRUSTVALUEUPDATEVERBOSE 0
#define MALICIOUSDROPVERBOSE 0

#endif // TRUSTCONSTANTS_H
```