**An evaluation of reputation spreading in mobile ad-hoc networks**

**(HS-IKI-EA-04-204)**

**Martin Håkansson**
**(e01marha@student.his.se)**

*School of Humanities and Informatics*
*University of Skövde, Box 408*
*S-54128 Skövde, SWEDEN*

Final Year Project in Software Engineering, Spring 2004.
Supervisor: Henrik Grimm
Supervisors at Ericsson: Per Gustavsson and Christoffer Brax

Examiner: Björn Lundell

**An evaluation of reputation spreading in mobile ad-hoc networks**

Submitted by Martin Håkansson to Högskolan Skövde as a dissertation for the degree of B.Sc., in the school of Humanities and Informatics.

**2004–09–07**

I certify that all material in this dissertation which is not my own work has been identified and that no material included for which a degree has previously been conferred on me.

Signed:_____

# An evaluation of reputation spreading in ad-hoc networks

**Martin Håkansson (e01marha@student.his.se)**

**Abstract**

The use of mobile ad-hoc networks (MANETs) is growing. The issue of security in MANETs is not trivial, since such networks have no fixed infrastructure and therefor centralised security is not applicable. MANETs are also more sensitive to attacks due to their wireless communication channels and their spontaneous nature.

All kind of cooperation requires a sense of trust. The opinion about trust in other entities can be used as a mean to dynamically allow for secure cooperation in MANETs, as soft security. And also to counter some of the inherited security problems of MANETs.

To use opinions as a security paradigm in MANETs the opinions about other nodes has to be spread as reputation about a node. This reputation spreading can be done through spreading of opinions or the spreading of evidences about a nodes behaviour.

In this work evidence and reputation spreading are compared to each other. This comparison shows that they are quite similar from a security point of view but that they differ in scalability.

**Keywords:** MANET, Ad hoc network, Trust, Opinion, Security, scalability.

# Contents

# List of Figures

# 1 Introduction

Here in an introduction to distributed systems and ad-hoc networks is given, followed by an introduction to trust and security.

According to Coulouris et al. (2001) distributed systems are everywhere. The Internet is one form of distributed system that most are familiar with. But there are many other distributed networks that is also a part of our everyday life such as the mobile phone networks and the ATM systems. Coulouris et al. (2001) state the main motivation for constructing a distributed system as the desire to share resources. The term 'resource' is quite abstract in this context as it includes all things that can be usefully shared in a networked computer system, both hardware components and software-defined entities.

Coulouris et al. (2001, p. 1) define a distributed system as "one in which components located at networked computers communicate and coordinate their actions only by passing messages". They identify the following three characteristics of a distributed system: *a*) concurrency of components, *b*) lack of global clock, and *c*) independent failure of components.

They mention mobile computing as one use of distributed systems. They define mobile computing as "the performance of computing tasks while the user is on the move, or visiting places other than their usual environment" (Coulouris et al. 2001, p. 6). They state that technological advancements in device miniaturization together with wireless networking has led to an increased integration of portable computing devises into distributed systems. Mobile computing allows users to access their resources via the devices they carry with them. They also mention an increasing provision for the users of mobile computing to utilise resources that are conveniently nearby, which is called *location-aware computing*.

The combination of location-aware computing and mobile computing can utilise a spontaneous way to connect to nearby resources, called *ad-hoc networking* (or *spontaneous networking*).

Zhou and Haas (1999) present ad-hoc networking as a paradigm of wireless communication for mobile hosts. Furthermore they mean that in an ad-hoc network there is no fixed infrastructure, but that mobile hosts which are within eachothers range communicate directly via wireless links. Hosts which can not reach eachother directly depends on other hosts to act as routers. The mobility of the nodes and the independence of a fixed infrastructure make it possible for ad-hoc networks to form and break down, into subnets or individual hosts, 'any time any where'. In addition Eschenauer et al. (2002) identify some additional characteristics of a

*mobile ad-hoc network (MANET)* as; *a*) lack of a fixed networking infrastructure, *b*) high mobility of the nodes, *c*) limited-range, and *d*) unreliability of wireless links.

The sharing of resources in distributed systems is a form of cooperation between the hosts in the network. Gambetta (2000) states that cooperation requires trust in others as well as the belief that one is trusted, or *accepted* by others. He also identify the will to cooperate as the motivation for trust. Hence to enable cooperation in any distributed network, hosts have to be able to trust eachother. In a fixed network it is convenient to state which hosts to trust and which not to trust in a static manner. But in ad-hoc networking and MANETs this is not the case. Due to the spontaneous nature of the network we can not predict which hosts will join the network and at what time.

Jøsang and Knapskog (1998) and Jøsang (1998) present a formal method for representing and valuating trust, as opinions, based on evidences of the good or bad nature of an entity. They also present a way to reach a conjunctive opinion about an entity based on the consensus of several entities' opinions.

According to Lamsal (2001) it will be hard, if not impossible, to implement proper security in an ad-hoc network environment without the concept of trust. Hard security is a paradigm where an entity has either blind trust or complete distrust in another entity (Abdul-Rahman and Hailes 1997). This is called *hard trust*. In *soft security* there is a scale of trust, so that an entity can trust another entity to a certain degree. This is called *trust*. Trust with a sense of uncertainty, that is a concept of how good the trust value is, is called an *opinion*.

# 2 Background

Herein the problems with security in networks in general and more specific security in MANETs is presented. Followed by a description of soft security, and trust and opinions as a mean of achieving soft security.

## 2.1 Security

*Security* is the protection of ones property. Specifically, in the case of computer security it is ones data. Security is to ensure that data is accessed only by those who should have access to the data.Venkatraman and Agrawal (2000) lists the classification of security services in any network as follows:

**Confidentiality:** Information in a computer system and transmitted information is readable only to those who are authorised to read the data.

**Encryption:** The origin of a message can be identified, with an assurance that the identity is correct.

**Integrity:** Information can only be modified by those authorised to modify data. Modification includes writing, changing status, deleting, creating or replaying transmitted messages.

**Access control:** Access to information can be controlled by or for the target system.

**Availability:** The information is accessible to those who are authorised to access the information.

There are several threats to computer security in any network environment. According to Coulouris et al. (2001) these threats all fall into one of the following three classes:

**Leakage:** Unauthorised recipients acquire information.

**Tampering:** Unauthorised alteration of information.

**Vandalism:** Interference with the operation of the system without gain to the perpetrator.

They also classify the methods of attack towards a network as:

**Eavesdropping:** Obtaining copies of messages without authorisation.

**Masquerading:** Using the identity of another principal, to receive or send messages, without their authorisation.

**Message tampering:** Intercepting and altering the contents of messages before passing them on to the intended recipient.

**Replaying:** Storing intercepted messages and sending them at a later date.

**Denial of service:** Flooding the network or a resource in the network with messages in order to deny access for others.

These are theoretical threats to a network. According to Coulouris et al. (2001) a successful attack depend upon the discovery of loopholes in the security of systems, and these are common in todays systems.

These security threats all apply to hard security as well as soft security.

## 2.2 Security in ad-hoc networks

Zhou & Haas (1999) states that the special features of MANETs, pose both opportunities and challenges in achieving security.

The wireless communication links makes the network susceptible to link attacks, as the links are relatively easy to eavesdrop and also to send messages on from an outside source. Zhou & Haas (1999) identifies some ways this might be used to actively attack the network by deleting messages, injecting erroneous messages, modify messages, and to impersonate another node.

They also note that the mobility of nodes, often in a hostile environment, makes it necessary to protect the network, not only from attacks from the outside, but also from within the network. Thus, they state that, MANETs should have a decentralised security architecture with no central entities.

Zhou & Haas (1999) also present the spontaneous nature of MANETs as a problem for security. As nodes can leave and join the network in a spontaneous manner it is desirable, if not necessary, for the security mechanisms to adapt to these changes in the network.

One way of handling the problem with the spontaneous nature of MANETs as well as the possibility of attacks from within the network is to use a soft security mechanism. Soft security is described in the following section.

## 2.3 Soft security

Soft security is when access can be given on a dynamic scale, as opposed to hard security where access is either given or not given to an entity. Soft security also offers a mean of dynamically changing the firmness of the security by being more restrictive in the access evaluation. One way of achieving soft security is to use trust, and opinions as shown in the following sections.

### 2.3.1 Trust

Trust is a natural part of human life. Everyday we evaluate the trustworthiness of many different things. When we watch TV or search the Internet for information we always evaluate the trustworthiness of the source of the information, and most of the time we don't think about it, we just do it. For example, we use the telephone trusting that our conversation is not intercepted by anyone other then the one it's intended for.

Since these evaluations are most often made without us actively thinking of them, most of us would not be able to answer why they trust or distrust some information. Also most of us would not be able to answer the question 'what is trust?'.

Rundell (1995) defines trust as "to believe that someone is honest and will not harm you". This is a very basic definition of what trust is and needs to be expanded to be useful in a computer security context. Gambetta (2000, p.217) defines trust as:

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

So if we have a concept of trust, we also by association have a concept of distrust, as the opposite of trust.

Whether or not we trust an entity to perform a given action can be though of as a binary statement, it is either true or false. But due to our lack of knowledge it is impossible to to be certain weather it is true or false, and therefore there can only be an *opinion* about it (Jøsang & Knapskog 1998).

An opinion is based on the previous known behaviour of an entity. The previous behaviour is known through *evidences* which are positive or

negative observations of the entity's behaviour. Evidences are based on an evaluation of the behaviour of the entity to a given event (Jøsang & Knapskog 1998).

### 2.3.2 Trust classes

Yahalom et al. (1993) introduces the concept of trust classification, as they see the traditional security classification where an entity is either considered trusted or untrusted as oversimplified. They state that in a real world setting it would be reasonable to have different degrees of trust in respect to different tasks, regarding one single entity. They also identify a number of typical tasks, or *trust classes*, which are further described by Beth et al. (1994).

Beth et al. (1994, p. 5) states that trust with respect to one trust class is independent of trust with respect to other trust classes. This is true in general, but there might be a global opinion about an entity as being bad or good. In a real world setting some people are regarded by us to be generally more or less trustworthy. This affects, in conjunction with the trust regarding a specific task, our trust in them to carry out that task. This implies that there is a potential dependence between trust with respect to different trust classes.

### 2.3.3 Opinions

The absence of trust should be handled separately as a grade of uncertainty in the set of evidences on which the opinion is based, or the *evidence base*.

Jøsang & Knapskog (1998) and Jøsang (1998) presents a way of representing opinions where opinions are modelled as three respectively dependent values between 0 and 1, namely belief, disbelief, and uncertainty. This method offers a way to differentiate between an irrational entity, one who is neither trusted nor distrusted, and an entity about which there is not enough knowledge to make a classification. This makes it possible to elaborate on whether or not it's necessary to try and find further evidence about the behaviour of an entity.

This opinion translates into degrees of belief or disbelief and the lack of knowledge into degrees of uncertainty. Jøsang & Knapskog (1998) presents a way to express this mathematically as:

$$b + d + u = 1, \quad \{b, d, u\} \in [0, 1]^3 \tag{1}$$

Where $b, d$, and $u$ is belief, disbelief, and uncertainty respectively. This can be illustrated as a point $\{b, d, u\}$ in the the opinion triangle, seen in

figure 1.

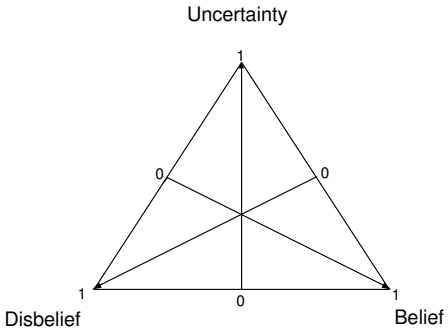


Figure 1: Opinion triangle. (From Jøsang & Knapskog, 1998).

Jøsang & Knapskog (1998) also show how an opinion can be derived from evidence. Let $r_p$ be the number of positive evidences about event $p$ and let $s_p$ be the number of negative evidences for the same event. Then the opinion about event $p$ denoted as $\omega_p = \{b_p, d_p, u_p\}$ is defined by equation 2.

$$\omega_p = \begin{cases} b_p = \frac{r_p}{r_p+s_p+1} \\ d_p = \frac{s_p}{r_p+s_p+1} \\ u_p = \frac{1}{r_p+s_p+1} \end{cases} \tag{2}$$

This can be used to derive an opinion from a set of evidences about an entity's behaviour in an event.

Jøsang & Knapskog (1998) also shows that a conjunction between two such opinions, as the opinion about both events, is defined by equation 3.

$$\omega_{p\wedge q} = \begin{cases} b_{p\wedge q} = b_p b_q \\ d_{p\wedge q} = d_p + d_q - d_p d_q \\ u_{p\wedge q} = b_p u_q + u_p b_q + u_p u_q \end{cases} \tag{3}$$

Where $\omega_{p\wedge q}$ is the conjunction between the opinions $\omega_p$ and $\omega_q$. The conjunction is a generalisation of the logical binary AND. If opinions of blind trust or complete distrust is conjuncted it will produce the truth table of the logical AND.

### 2.3.4 Consensus between opinions

Jøsang & Knapskog (1998) identifies and defines three types of consensus. *a*) The one between *independent opinions*, *b*) the one between *fully dependent opinions*, and *c*) the one between *partially dependent opinions*. To decide which definition applies to a given situation, the evidence

founding the opinions has to be examined. If the opinions are based on evidence based on the same events, then the opinions are dependent. If no such evidence exists the opinions are independent. If some of the evidences are based on the same events and some are not, then the opinions are partially dependent.

Here only the consensus between independent opinions will be defined.

Jøsang & Knapskog (1998) shows that to get a consensus between two independent opinions the evidence can be added up as shown in equation 4.

$$\begin{aligned} r_p^{A,B} &= r_p^A + r_p^B \\ s_p^{A,B} &= s_p^A + s_p^B \end{aligned} \tag{4}$$

Where $r_p^A$ and $s_p^B$ is entity $A$s' evidence base for event $p$, and $r_p^B$ and $s_p^B$ is entity $B$s' evidence base for the same event.

An equivalent operator for opinions is defined by equation 5, derived from equation 4 and equation 2.

$$\omega_p^{A,B} = \begin{cases} b_p^{A,B} = (b_p^A u_p^B + b_p^B u_p^A)/K \\ d_p^{A,B} = (d_p^A u_p^b + d_p^B u_p^A)/K \\ u_p^{A,B} = (u_p^A u_p^B)/K \end{cases} \tag{5}$$

Where $\omega_p^{A,B}$ is the consensus between the opinions $\omega_p^A$ and $\omega_p^B$. The constant $K$ is defined as $K = u_p^A + u_p^B - u_p^A u_p^B$ (such that $K \neq 0$). If $K = 0$ both opinions have an uncertainty of 0, making it impossible to reach a consensus between those two opinions.

### 2.3.5 Reputation spreading

If an entity does not know whether to trust another entity or not (the trustee), the entity can ask a third party whom it trusts whether to trust the trustee or not (Lamsal 2001). This is called *reputation spreading*.

A reputation can be spread either as an opinion about an entity, *opinion spreading*, or as evidence about the behaviour of an entity, *evidence spreading*.

Abdul-Rahman & Hailes (1997) present a protocol for on-line trust distribution based on reputation. Their protocol has the goal to extend and generalise the current (at the time) approaches to security and trust management. The protocol defines a message structure for requesting reputation. The protocol also includes identification methods using encryption keys.

The protocol by Abdul-Rahman & Hailes (1997) is based on messages. If an entity wants a reputation about another entity it issues a

recommendation request message to receive a recommendation message. A given recommendation can be refreshed by issuing a refresh message. This is a rather basic protocol structure, but it shows how such a protocol could be built.

# 3 Problem description

Here in the aim of the work is presented and the objectives to reach that aim.

## 3.1 Aim

The aim of this work is to compare evidence spreading and opinion spreading in MANETs.

This comparison is to be done on scalability and security. Scalability is compared as the mobile and wireless nature of MANETs makes computational power and bandwidth an issue. Security is compared as reputation spreading is proposed to be a security paradigm.

## 3.2 Objectives

### 3.2.1 Compare scalability

Compare the paradigms in terms of scalability, both in bandwidth and in computational demand. The thesis is that evidence spreading will be more demanding on bandwidth but less so on computational demand, i.e. requires less CPU time. Due to that evidence spreading can first collect its evidences and then calculate an opinion form those evidences with one calculation, but opinion spreading requires each opinion to be added to a nodes opinion by consensus, so each opinion requested results in one calculation.

These two aspects of scalability is chosen since the mobile nature of MANETs makes computational power an issue as the mobility of the nodes limits their possibility to have high computational power. The wireless communications techniques used today has less bandwidth then traditional wire communication, thus keeping bandwidth usage down is essential in MANETs.

### 3.2.2 Compare security

Compare the paradigms in terms of security. The security aspects that are evaluated are availability and confidentiality. These aspects are the ones most applicable to reputation spreading, as other security methods would be needed to complement reputation spreading in the other security aspects.

Security in the context of soft security can be seen, not as a binary statement, but as a continuous scale. Thus there is no evaluation on weather or not the paradigms offer security or not. The question is not if it

is secure or not but if it is secure enough. That is dependent on the application of the system. The evaluation is which paradigm is safer or if they are roughly equal. The thesis is that evidence spreading is safer as it provides a way to separate dependent opinions from independent.

# 4 Method

Herein the choice of method is presented as well as a plan for how to execute the method, and analyse the results.

The choice of method to achieve the objectives is based on a review of existing work in the area. The existing work shows a lack of implementation and experimentation on the theories in the subject area. The background is also based on this review which is largely based on the literature review made by Lamsal (2001).

## 4.1 Experiment

To solve the objectives, in sections 3.2.1 and 3.2.2, an experimental approach has been chosen. The experiment is based on an implementation of the paradigms to be compared. As described by Berndtsson et al. (2002), an experiment focuses on the way in which a few variables are affected by the experimental conditions, or parameters. The parameters in this case is further described in the description of the experiment in section 5 as well as in appendix A.

As with all experiments there is the problem of validity and verifiability. To try to ensure the validity of the experiment presented here, a systematic development process of analysis, design, implementation, and testing has been undertaken. The analysis and design of which can be seen in section 5. The verifiability of the experiment is ensured by running the experiment several times with the same parameters and comparing the results to be equivalent.

There is also an issue of making the model relevant. To ensure the model is relevant a hypothetical scenario corresponding to the experiment is shown. The delimitations of the model in relation to the scenario are explained and motivated. These delimitations and their potential effect on the results are also taken into account in the conclusion and the discussion.

## 4.2 Data analysis

The data collected from the experiment is statistically analysed and visualised to allow for a conclusion based on the hypothesis. Statistical methods are used to show the possible difference in the scalability and security of the paradigms.

Basically the same experiments are executed for both paradigms, and the result of those runs are compared and analysed. The statistical methods used is based on the type of curves that are produced by the experiment.

The statistical methods used to compare plots is visual comparison of the graphs and mean value of the samples. Mean value of samples according to Zar (1999, p. 123), is equivalent to a two-sampled t-test, used for curve comparison.

# 5 Experiment

Herein the experiment is explained. Starting with a description of the scenario, followed by definitions and descriptions of the experiment implementation. Also the delimitations of the experiment implementation is listed and motivated, and some technical details of the implementation is given.

## 5.1 Scenario

The experiment is based on a potential scenario where a given number of enemies, called *bad nodes*, are trying to get access to the network services by getting acceptance. The bad nodes try to achieve that by creating positive evidences about other bad nodes. The bad nodes also try to lower the defence and availability of the system by creating negative evidences about the nodes owning the network, called *good nodes*. This is to get good nodes to be classed as rejected and thus not be able to spread their negative evidence about the bad nodes. This corresponds to a potential denial of service attack to a system, where an enemy is trying to prevent access by good nodes, as well as an attempt to compromise confidentiality by gaining access to the system.

## 5.2 Reputation spreading

An entity can spread either its own opinion about another entity, called *opinion spreading*, or it can spread the evidence on which it bases its opinion, called *evidence spreading* (see section 2.3.5). Both these methods serves to spread information about the reputation of an entity over the network. If the evidence is spread then they can be used to evaluate if the opinions are independent, partially dependent, or fully dependent.

## 5.3 Acceptance test

In order to give access to an entity in a system there must be a sense of trust or acceptance of that entity.

*Acceptance* can be viewed as a subset of the set of opinions that can be had about an entity. The subset can be defined as a sector with a given radius, in the opinion triangle, from blind belief, $\omega_{blindbelief} = \{1, 0, 0\}$. We define acceptance as seen in formula 6, derived from the distance formula.

$$acceptance_p \Leftrightarrow \sqrt{(b_p - 1)^2 + (d_p - 0)^2 + (u_p - 0)^2} < AL \tag{6}$$

Where $acceptance_p$ is the acceptance of an entity to perform task $p$. The acceptance can be used to give an entity access. The value $AL$ (acceptance limit) is set independently for each trust class. The area in the opinion triangle (see figure 1) defined by formula 6 can be seen as the gray area in figure 2.

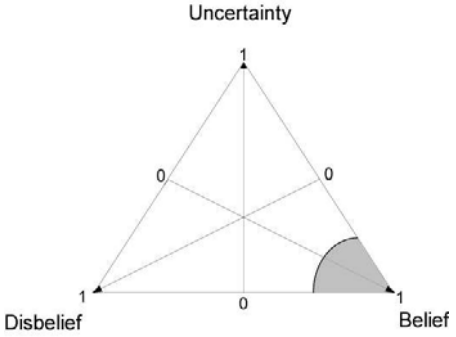


Figure 2: Acceptance as an area in the opinion triangle.

According to the definition by Gambetta (2000) a notion of trust requires a notion of distrust, or rejectance. Rejectance is not the same as the lack of acceptance. We define *rejectance* as enough certainty to know not to accept an entity. The definition of rejectance can be seen in formula 7.

$$rejectance_p \Leftrightarrow \neg acceptance_p \ \wedge \ u_p < RL \tag{7}$$

Thus rejectance can be separated from the lack of acceptance. The constant $RL$ (rejectance limit) is set independently for each trust class. The area in the opinion triangle defined by formula 7 can be seen as the gray area in figure 3.

Given the formulas for acceptance and rejectance. Unclassified opinions are opinions that are neither accepted nor rejected, calculated with formula 8. Unclassified opinions can be seen as the gray area in figure 4.

$$unclassified_p \Leftrightarrow \neg(acceptance_p \ \vee \ rejectance_p) \tag{8}$$

Note that every possible opinion is a member of exactly one of $acceptance_p$, $rejectance_p$, or $unclassified_p$.

## 5.4 On-line trust protocol adaptation

The ideas of a protocol by Abdul-Rahman & Hailes (1997), presented in section 2.3.5, is used as a base for the protocol used in the experiment.

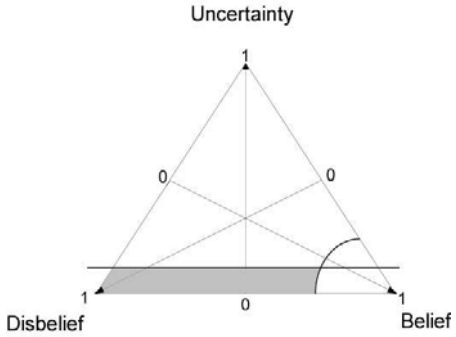


Figure 3: Rejectance as an area in the opinion triangle.

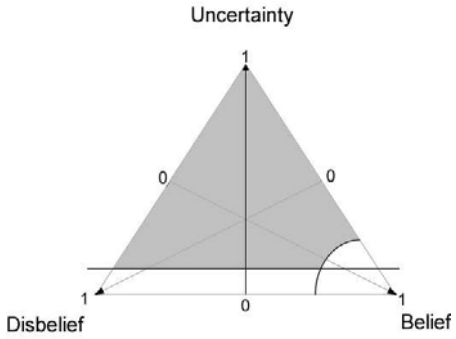


Figure 4: Unclassified opinions as an area in the opinion triangle.

Since there is no trust classes in the experiment, the only type of evidence request that is implemented is a request for all evidences about a given entity.

A request for opinions about an entity is also added, this is virtually the same request as the one for evidence, but instead of returning evidences it returns the opinion about the entity. An opinion request always returns exactly one opinion.

The protocol data structures presented as pseudo code follows:

**Request**

*node id* ← The id of the node making the request.
*requested id* ← The id of the node about which the request is for.
*live* ← Lifetime for the request in number of hops.
*history* ← List of nodes who already got the request.

A request is the same independent of the paradigm used. A request is initialised as a request structure with *node id* and *requested id* set to the

corresponding values and the live variable set to the number of hops, i.e the number of sub levels, the request should live. *History* is initialised as a list containing the requesting entity and updated with every node that handles the request. This is done to enable the request to not be handled by any node more than once.

**Evidence**

*evidence id* ← The id of the evidence.
*node id* ← Id of the node who created the evidence.
*positive* ← True if the evidence is positive, false if it is negative.
*object id* ← Id of the node the evidence is about.
*expiery* ← Expiery time for the evidence.

A request will, if the paradigm of the experiment is evidence, result in a list of evidences. The *evidence id* is together with the *node id* a unique identifier for the evidence. This makes it possible to ensure that an opinion based on evidences only uses every evidence once. The *expiery* is the time at which the evidence expires and is removed.

**Opinion**

*node id* ← Id of the node which created the opinion.
*object id* ← Id of the node which the opinion is about.
*opinion* ← The opinion about node with *id object id*.
*expiery* ← The expiery time for the opinion.

A request will, if the paradigm of the experiment is opinion, result in an opinion representing the consensus of the opinions found by the request. The *expiery* of the opinion is set by the receiving node itself and represents how long it will trust the opinion to be correct.

## 5.5 Delimitation

The experiment does not change the behaviour of its nodes. A bad node is always bad and a good node is always good. In a real scenario bad nodes would most probable appear to be good nodes and then start to act bad when they think that they have got a status as accepted. This is a likely scenario as it would improve the damage a bad node could make.

The experiment views all evidences as independent. Opinion spreading in the experiment is handled as independent opinions (see section 2.3.4). To handle dependent opinions the entire evidence base for the opinion has to be communicated, which would make the method equivalent to evidence spreading with the extra overhead of sending an opinion as well as the

evidences. The equivalence of evidence spreading and opinion spreading when considering dependency between opinion is proved by Jøsang & Knapskog (1998).

## 5.6 Parameters

The parameters that can be set for the experiment follows:

**Number of nodes** The total number of nodes in the experiment, including both good and bad.

**Number of bad nodes** The number of bad nodes out of the total number of nodes. Must be less than or equal to the number of nodes.

**Minimum and maximum number of connections** The number of nodes that each node is connected to is between the minimum and maximum number of connections. The number of connections are randomised for each node. Which nodes a node connects to is also randomised.

**Number of nodes to examine** The number of nodes each node evaluates. Each nodes evaluates all nodes it is connected to and, potentially, some more. The number of nodes to examine must be greater than or equal to the maximum number of connections.

**Number of passes** The number of passes that the experiment runs for. For each pass, every node evaluates the nodes it should examine

**Chance of evidence creation** The probability that a node creates an evidence about another node that it is connected to in a given pass.

**Evidence lifetime** The lifetime for an evidence as a number of passes from its creation.

**Opinion lifetime** The lifetime for an opinion, spread through opinion spreading, as a number of passes from its creation. This parameter is only applicable when the paradigm is set to opinion spreading.

**Connection lifetime** The lifetime for the connections as a number of passes. The connections are re-randomised when the connection lifetime expires, to simulate ad hoc networks.

**Request timeout** The number of hops for a request to live. A hop is one level calling its neighbour nodes, so if node A is connected to node B and node B is connected to node C then a request from A to B would be one hop and a request from A to C through B would be 2 hops.

**Acceptance limit** The acceptance limit for a node as seen in formula 6.

**Rejectance limit** The rejectance limit for a node as seen in formula 7.

**Inner acceptance limit** The inner acceptance limit is a trust limit equal to or less then the acceptance limit. This is used to evaluate if an opinion is near unclassified, to allow a node to request evidences before a node gets unclassified.

**Inner rejectance limit** The inner rejectance limit, less then or equal to the rejectance limit. Used to evaluate if an opinion is near unclassified.

**Paradigm** The paradigm to use for spreading of reputation. Either evidence spreading or opinion spreading.

## 5.7 Design

Implementation details, in the form of software design and code, can be found in appendixes A and B. Here pseudo code for the reputation spreading algorithms is given as well as the flow of the experiment execution.

Pseudocode for the evidence collecting algorithm:

**Require:** $n \leftarrow$ the node running the algorithm.
**Require:** $e \leftarrow$ the node to examine.
**Require:** $opinion_e \leftarrow$ n's current opinion about $e$.
  **if** $opinion_e \in$ near uncertain **then**
    **for all** nodes $a \in$ connected to $n$ **do**
      **if** $a \notin$ distrusted **then**
        ask $a$ for evidence about $e$
      **end if**
    **end for**

recalculate $opinion_e$ based on evidence
**end if**

Pseudocode for the opinion collecting algorithm:

**Require:** $n \leftarrow$ the node running the algorithm.
**Require:** $e \leftarrow$ the node to examine.
**Require:** $opinion_e \leftarrow$ n's current opinion about $e$.
**if** $opinion_e \in$ near uncertain $\wedge$ $opinion_e$ old **then**
**for all** nodes $a \in$ connected to $n$ **do**
**if** $a \notin$ distrusted **then**
ask $a$ for opinion about $e$
$opinion_e \leftarrow$ consensus between $a$'s opinion and $opinion_e$
**end if**
**end for**
**end if**

A flowchart for the algorithms are shown in figure 5.

The flowchart shown in figure 6 shows how the experiment is run in passes. The run pass for node $n$ means to run the evidence collecting or opinion collecting algorithm for node $n$.

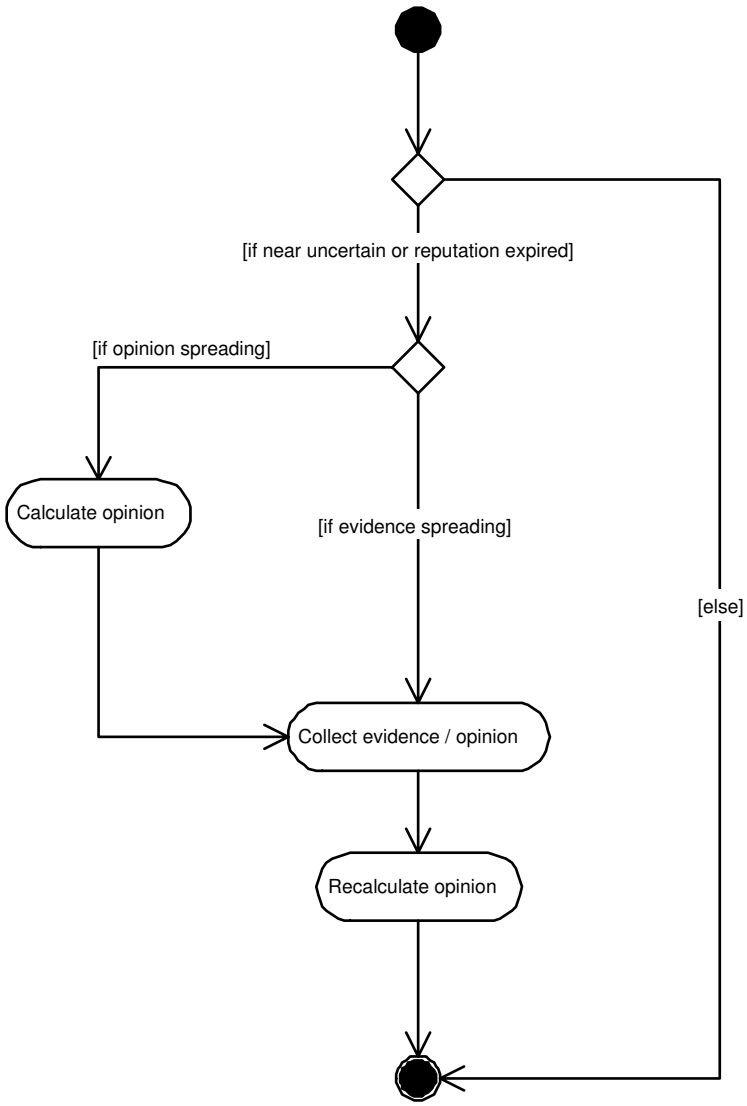


Figure 5: Flowchart for evidence and opinion collecting algorithms.

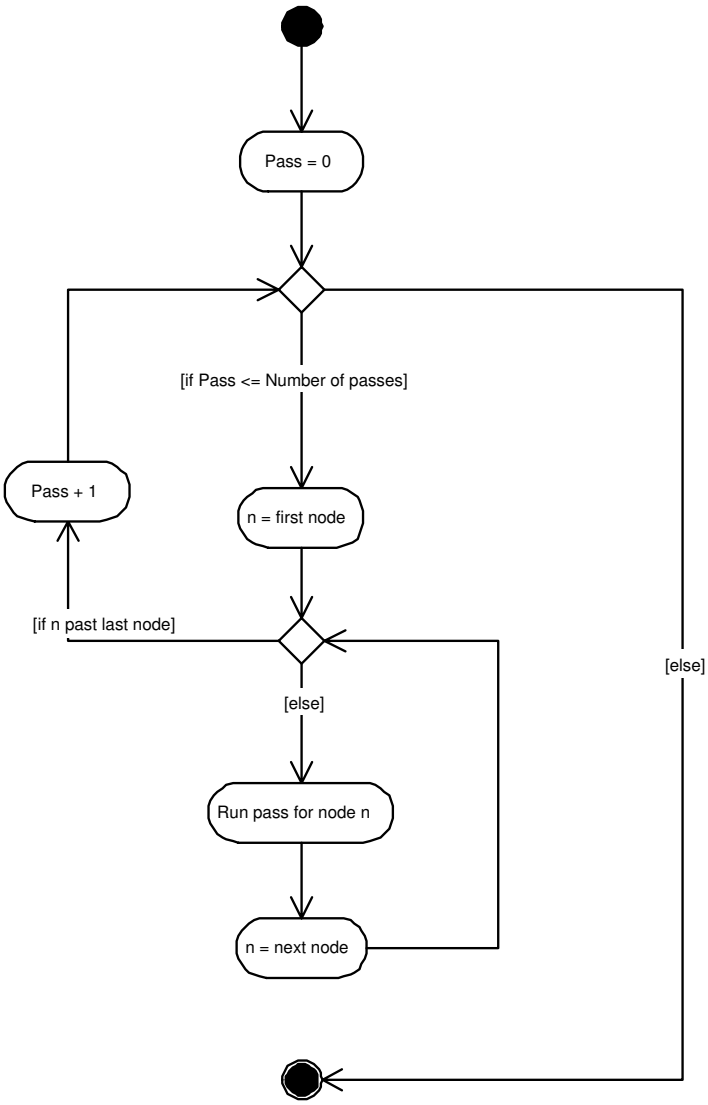


Figure 6: Flowchart for the experiment.

# 6 Results and analysis

Herein the results of the experiment is given and as well as some analysis of those results.

The default parameters for the experiments are shown in table 1. The parameters changed for a given experiment are given in separate tables.

Table 1: Default parameter settings.

| **Parameter** | **Value** |
|---|---|
| Number of nodes | 100 |
| Number of bad nodes | 0 |
| Minimum number of connections | 10 |
| Maximum number of connections | 15 |
| Number to examine | 20 |
| Number of passes | 100 |
| Chance of evidence creation | 0.03 |
| Chance of evidence false | 0.001 |
| Evidence lifetime | 15 |
| Opinion lifetime | 10 |
| Connection lifetime | 50 |
| Request timeout | 2 |
| Acceptance limit | 0.3 |
| Rejectance limit | 0.1 |
| Inner acceptance limit | 0.3 |
| Inner rejectance limit | 0.1 |
| Paradigm | Evidence and opinion |

## 6.1 Scalability

Scalability of calculation is measured as the number of times an opinion is calculated, either from evidences or by consensus, as well as the number of times acceptance is calculated. Data sent is the number of evidences or opinions sent. The size of the structure sent has not been taken into account.

Table 2: Parameter settings to figures 7, 8, and 9.

| **Parameter** | **Value** |
|---|---|
| Number of bad nodes | 0 |
| Number to examine | 10 increasing to 55 |
| Minimum number of connections | 5 |
| Maximum number of connections | 10 |
| Evidence lifetime | 20 |
| Opinion lifetime | 5 |
| Chance of evidence false | 0.003 |
| Inner acceptance limit | 0.25 |
| Rejectance limit | 0.15 |

A scalability test with no bad nodes shows that evidence spreading scales much better on calculations, as seen in figure 7. The mean value of

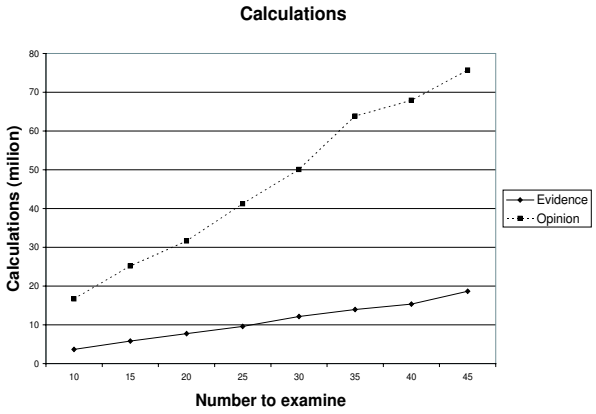


Figure 7: Calculations with increasing number of nodes to examine, and no bad nodes.

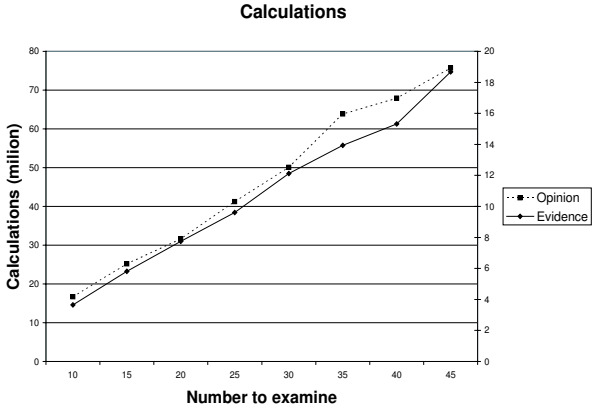


Figure 8: Calculations, same as figure 7 but with dual Y axis.

the samples is 12 million to 55 million, which is substantially lower. Placing the two plots in the same graph on different Y axis, as seen in figure 8, shows that they both scale linearly. However opinion spreading has a much steeper linear scalability. This is due to the fact that the evidence collecting algorithm allows for all evidences to be collected before an opinion calculation is performed, while the opinion collecting algorithm requires consensus to be reached for every collected opinion, see section 5.7.

The number of data packets sent does not show as big difference, as seen in figure 9. The mean value of the samples also show a slight difference, 23 million for evidence spreading to 17 million to opinion spreading.

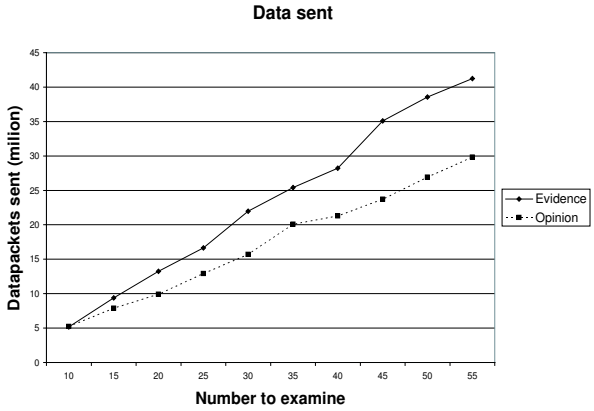


Figure 9: Data sent with increasing number of nodes to examine, and no bad nodes.

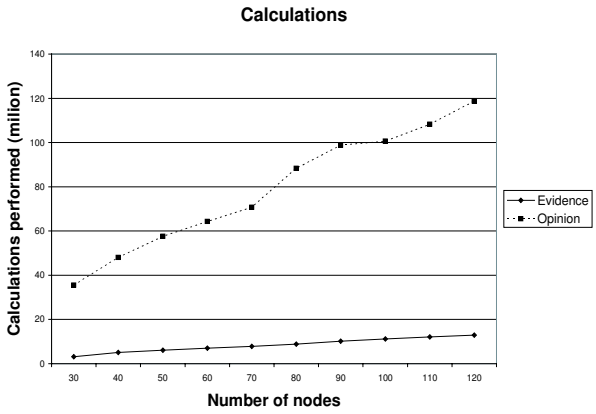


Figure 10: Calculations with a growing number of nodes, and no bad nodes.

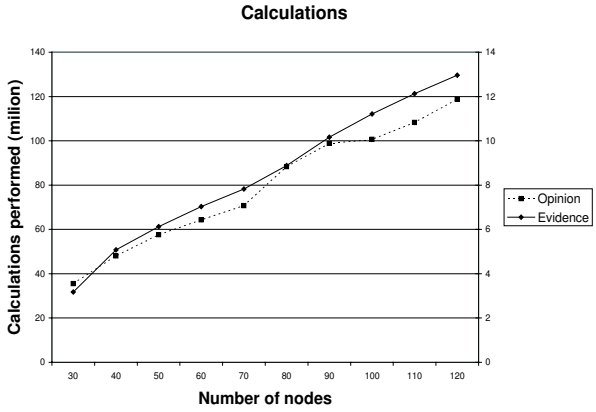


Figure 11: Calculations, same as 10 but with dual Y axis.

Table 3: Parameter settings to figures 10, 11, and 12.

| **Parameter** | **Value** |
|---|---|
| Number of nodes | 30 increasing to 120 |
| Number of bad nodes | 0 |

Varying the number of nodes in the system and still having no bad nodes, gives about the same scalability as varying the number of nodes to examine, as seen in figure 7 and 10, and 9 and 12.

Evidence spreading scales much better on calculations than reputation spreading, as seen in figure 10, sample mean is 8 to 79 million. Both scale

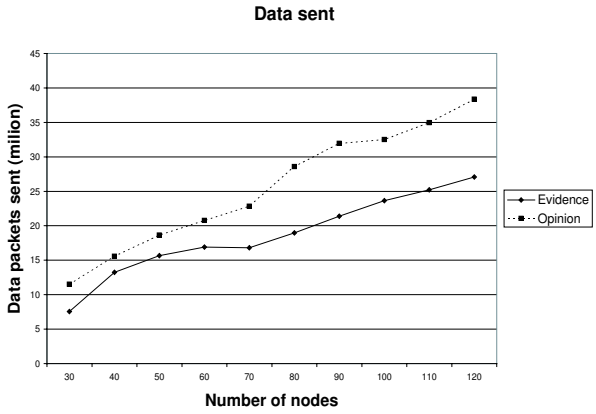


Figure 12: Data sent with a growing number of nodes, and no bad nodes.

linearly, as seen in figure 11, but reputation spreading scales much steeper as seen in figure 10.

Data packets sent shows slightly better for evidence spreading as seen in figure 12. The sample means are 19 million for evidence spreading and 26 million for opinion spreading. The difference is not big enough to prove that one is better than the other for data sent scalability.

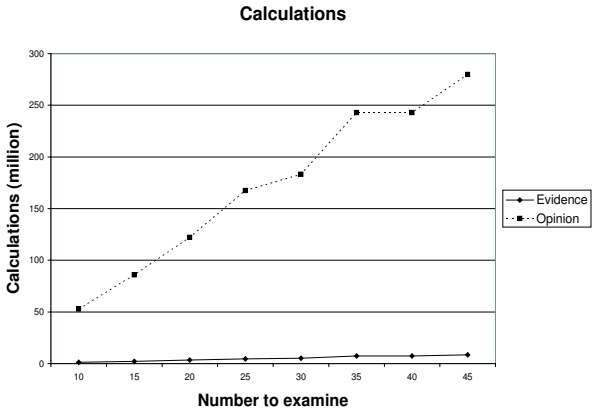


Figure 13: Calculations with growing number of nodes to examine and 20% bad nodes.

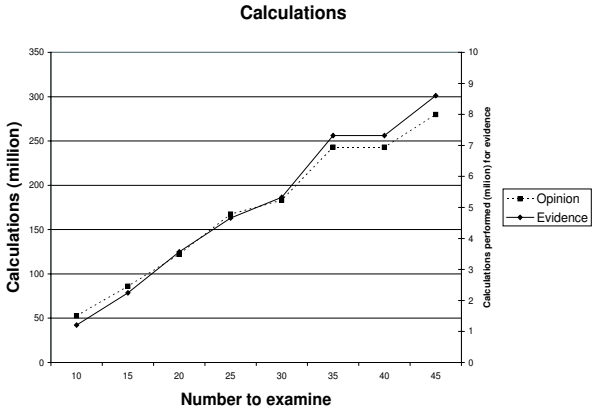


Figure 14: Figure 13, with dual Y axis.

Table 4: Parameter settings to figures 13, 14, and 15.

| **Parameter** | **Value** |
|---|---|
| Number of bad nodes | 20% of nodes |
| Number to examine | 10 increasing to 45 |
| Minimum number of connections | 5 |
| Maximum number of connections | 10 |

Adding 20% bad nodes (see table 4), and rerunning the experiment described in table 2. We can see that the addition of bad nodes does not affect the relative scalability in this case, as seen in figures 7 and 13, and figures 9 and 15.

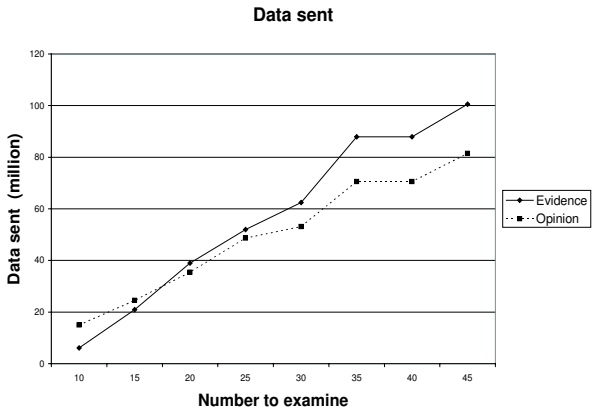


Figure 15: Data sent with growing number of nodes to examine and 20% bad nodes.

Evidence spreading has better scalability for calculations, as seen in figure 13, the sample mean values are 5 to 72 million.

Scalability in data sending is similar for both paradigms, as seen in figure 15, the sample mean values are 57 million for evidence spreading to 50 million for opinion spreading.

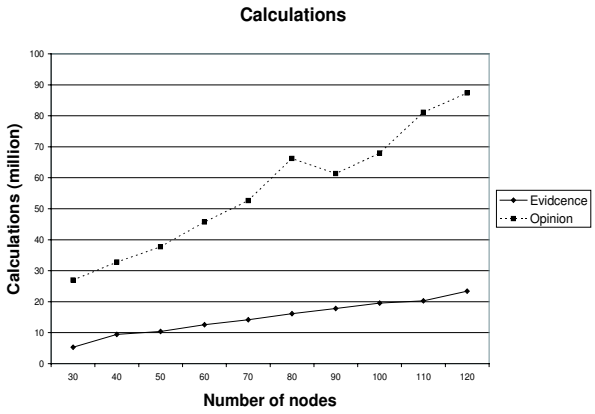


Figure 16: Calculations with growing number of nodes, and 20% bad nodes.

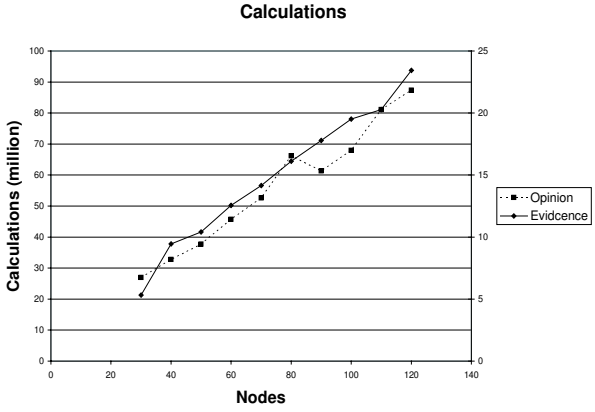


Figure 17: Figure 16 with an added Y axis.

Table 5: Parameter settings to figures 16, 17, 18, and 19.

| **Parameter** | **Value** |
|---|---|
| Number of nodes | 30 increasing to 120 |
| Number of bad nodes | 20% of nodes |

Adding 20% bad nodes to the experiment described in table 3, as seen in table 5. Shows that bad nodes affects the data sending scalability so that opinion spreading scales better than evidence spreading, as seen in figure 18. The sample mean values are 50 to 18 million, which shows that opinion spreading in this case scales better. They both scale linearly as seen in figure 19 but opinion spreading has a steeper angle.

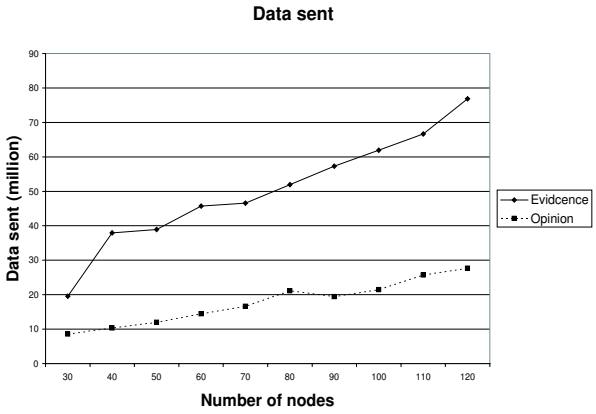


Figure 18: Data sent with growing number of nodes, and 20% bad nodes.

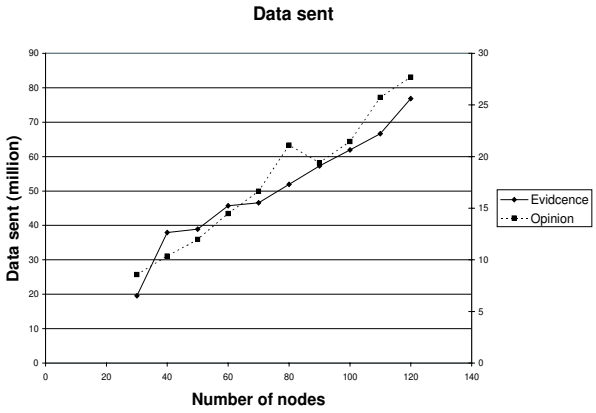


Figure 19: Figure 18 with an added Y axis.

On calculations evidence spreading still scales better, as seen in figure 16. The sample mean values are 15 to 56 million. They both scale linearly as seen in figure 17.

## 6.2 Security

Availability is measured as the percentage of good nodes that are accepted into the system. Confidentiality are measured as the percentage of bad nodes not accepted into the system.

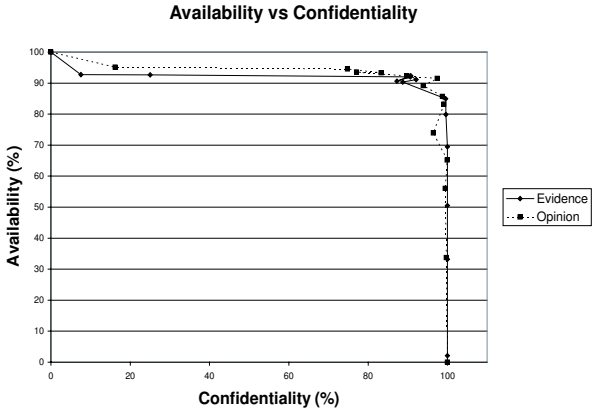


Figure 20: Availability vs security with 10% bad nodes.

Table 6: Parameter settings to figure 20.

| **Parameter** | **Value** |
|---|---|
| Number of bad nodes | 10% of nodes |
| Acceptance limit | 1.5 declining to 0.0 |
| Rejectance limit | 0.5 declining to 0.0 |
| Inner acceptance limit | Acceptance limit |
| Inner rejectance limit | Rejectance limit |

The diagram in figure 20 shows the confidentiality versus the availability of both paradigms. As can be seen evidence spreading has more availability on 100% confidentiality. Opinion spreading does not reach 100% confidentiality until its availability drops to 0%. This means that in this case evidence allows for a higher availability with total confidentiality, even though the mean availability is slightly higher for opinion spreading, 76% to 71% for evidence spreading. The mean confidentiality is also slightly higher for opinion spreading, 82% to 79% for evidence spreading. As evidence spreading has the higher availability for 100% confidentiality, 69% to 0% for opinion spreading.

The mean values have too small a difference to draw a conclusion from them. But the fact that evidence spreading reaches 100% security at such a relative high availability, is a small plus for evidence spreading.

Opinion spreadings higher mean value might be a result of opinion spreading calculating opinions with each evidence possibly used more than once, due to the lack of dependence checking. This makes the nodes in opinion spreading to be less uncertain about their opinions.

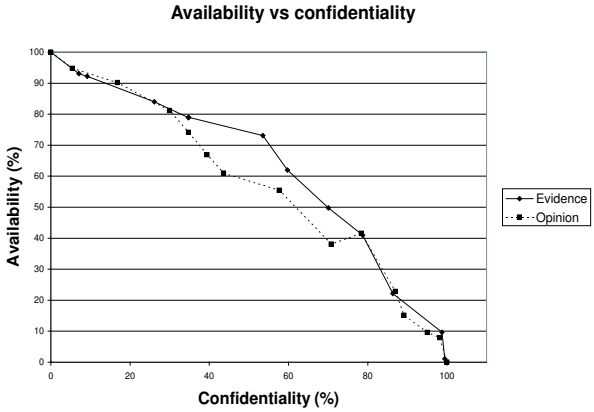


Figure 21: Availability vs security with 45% bad nodes.

Table 7: Parameter settings to figure 21.

| **Parameter** | **Value** |
|---|---|
| Number of bad nodes | 45% of nodes |
| Acceptance limit | 1.5 declining to 0.0 |
| Rejectance limit | 0.5 declining to 0.0 |
| Inner acceptance limit | Acceptance limit |
| Inner rejectance limit | Rejectance limit |

The graph seen in figure 21 is the corresponding graph to figure 20 with 45% bad nodes instead of 10%. Opinion spreading has been slightly more affected by growth in number of bad nodes, which creates less quality in the evidence set. In this case evidence spreading gets to 100% confidentiality marginally before evidence spreading, evidence spreading reaches 100% confidentiality at 0.08% and opinion spreading reaches the same confidentiality level at 0%. This is too small a difference to draw any conclusion from it.

The mean availability is lower for opinion spreading, 50% to 52% for evidence spreading. The mean confidentiality is also lower, 56% to 57% for evidence. These differences are too small to draw any conclusions from them.

# 7 Related work

There are others who have presented theories and reports on trust and security. Some of them and their theories are presented here.

## 7.1 Lamsal

Lamsal (2001) presents a literature review on the subject of trust and security. He largely bases his review of trust on the theories presented by Beth et al. (1994). Lamsal (2001) presents some examples of security evaluation based on trust calculations. He also briefly discusses the growing need for trust as networks, in general, become more mobile and as ad-hoc networks become more common.

His work relates to this work in that it proposes the need for trust in securing MANETs, as well as providing an overview of the current state of the research in the area.

## 7.2 Beth, Klein et al.

Beth and Klein have together with Borcherding and Yahalom presented some rather extensive work on trust (see Beth et al, 1994 and Yahalom et al, 1993, 1994). They present a framework to evaluate the trust requirements of protocols. This is to say how much trust is needed for an entity to cooperate with another entity, in a given protocol. This aims to be able to compare protocols on their trust relationships and thus be able to design protocols so that their executions will reflect particular trust circumstances.

Their work relates to this work by being a way of expressing the, more or less, implicit trust relationships in protocols not directly using trust as a paradigm, as proposed in this work.

## 7.3 Eschenauer, Gligor, and Baras

Eschenauer et al. (2002) presents some differences of trust establishment in MANETs and on the Internet. Their work is based around military examples, but could be adopted for civilian use as well. They show that in MANETs, as opposed to (mobile) Internet, the trust establishment process has to be; *a*) peer-to-peer, *b*) short, fast, and on-line only, and *c*) flexible enough to allow uncertain and incomplete trust evidence.

They also propose the use of a swarm intelligence approach for the design of trust evidence distribution.

Their work relates closely to this work. As this work proposes a realisation of on-line trust, given that the evidence creation mechanisms can be expressed on-line, as well as incorporating uncertainty and incomplete trust evidence, as uncertainty and unclassified nodes.

## 7.4 Zhou and Haas

Zhou & Haas (1999) presents a way to secure MANETs and discusses the security issues of MANETs. They provide guidelines for secure routing in MANETs, by using multiple routes to tolerate failures in network communication. They also present a model for decentralised key management in MANETs and propose to use threshold cryptography to distribute trust among a set of servers.

Their work presents a way to secure MANETs however their solution has some weak points as it still relies on a set of security servers, even though not all have to be accessible, it still requires a number of them to be accessible. This makes their decentralised key management somewhat exposed to attack, by attacking the security system it self as it is not entirely distributed among the nodes.

# 8 Conclusion and future work

Here the conclusion of this work is presented along with a discussion on the subject. Followed by a discussion on future work in the area.

## 8.1 Conclusion

In scalability opinion spreading shows a clear advantage in the number of calculations needed to be performed. Data sent only showed a difference with 20% bad nodes and a growing number of nodes in the network. So if an attack from bad nodes is expected and the number of nodes in the scenario is large then evidence spreadings larger bandwidth use might be an issue. Otherwise evidence spreadings lower computational need puts evidence spreading in favour.

When it comes to security the experiments does not show any conclusive difference, except for evidence spreadings higher availability with total confidentiality at 10% bad nodes. This result shows that evidence is some what better if 100% confidentiality is vital, as could be the case in many military applications. The failure to show conclusive evidence for the thesis might be due to the lack of precise measurement. The differences might be too small to measure with such a small experiment. A greater experiment with more factors taken into account and a more thorough analysis might find a different result.

## 8.2 Discussion

Before implementing reputation spreading into security systems, more evaluation of reputation spreading should be conducted.

Opinions with reputation spreading could provide many advantages in cooperation with existing security applications. It could be used to aid system administrators in identifying problem users before they cause any real trouble. The use of opinion with reputation spreading however is dependent of a sound way of producing evidences, both positive and negative. This is a non trivial task in many scenarios as it requires a classification of actions as suspicious, for negative evidences, and, perhaps even harder, positive actions.

This work is a step towards such applications, but there is still a long way to go before reputation spreading can reach its full capacity as an aid in securing MANETs, as well as other computer based systems.

## 8.3 Future work

There is a need for further experiments on the aim of this work, to eliminate some of the delimitations of this work, as seen in section 5.5. It would also be interesting to implement reputation spreading in a real environment.

The protocols presented here could be evaluated and adapted to fit in the protocol suites used in MANETs today, such as current security protocols and routing protocols. They could also be optimised for use in such environments.

Trust as a security paradigm, could be compared to more traditional paradigms such as PKI, to evaluate the practical uses of trust as a security paradigm. Trust could also be evaluated as a complement to such paradigms.

Trusts ability to withstand other types of attacks such as masquerading is also an interesting field of study.

Trust could be used in other areas than security, such as information fusion, and e-commerce. A collected study of the theories in these areas and how they can be joined to create a greater understanding and how the same theories can be used in different areas would be useful.

The issue of evidence creation, both on-line and off-line, needs to be researched further to be reliable enough to use opinions in a useful manner.

# 9 Acknowledgement

I want to thank EMW in Skövde and all its personnel for all support. Especially, my supervisors Christoffer Brax and Per Gustavsson. I also want to thank my examiner, Björn Lundell, and my supervisor at University of Skövde, Henrik Grimm, for all support and opinions. As well as Angelica Nordlund for help with the statistical methods and interpretations.

# 10 References

Abdul-Rahman A. & Hailes S. (1997). A Distributed Trust Model. pp. 48–60.

Berndtsson M. , Hansson J. , Olsson B. & Lundell B. (2002). *Planning and implementing your final year project with success!* Springer.

Beth T. , Borcherding M. & Klein B. (1994). Valuation of Trust in Open Networks. In *Proc. 3rd European Symposium on Research in Computer Security – ESORICS '94*, pp. 3–18.

Coulouris G. , Dollimore J. & Kindberg T. (2001). *Distributed systems concepts and design.* Pearson Education Ltd, third edn.

Eschenauer L. , Gligor V. & Baras J. (2002). On Trust Establishment in Mobile Ad-Hoc Networks.

Gambetta D. (2000). *Can we trust trust?*, chap. 13, pp. 213 – 237. Department of Sociology, University of Oxford, on-line edn. [On-line version] Available at Internet: http://www.sociology.ox.ac.uk/papers/gambetta213-237.pdf [Accessed 04.02.13].

Jøsang A. (1998). *Modelling trust in information security.* Ph.D. thesis, The Norwegian University of Science and Technology.

Jøsang A. & Knapskog S. J. (1998). A Metric for Trusted Systems. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pp. 16–29.

Lamsal P. (2001). Understanding trust and security. Available at Internet: http://www.cs.Helsinki.FI/u/lamsal/papers/UnderstandingTrustAndSecurity.pdf [Accessed 04.02.15].

Rundell M. (1995). *Longman dictionary of contemporary english.* Longman Group Ltd, new edn.

Venkatraman L. & Agrawal D. P. (2000). A novel authentication scheme for ad hoc networks. *Wireless Communications and Networking Confernce (WCNC 2000), IEEE* **3**:1268 – 1273.

Yahalom R. , Klein B. & Beth T. (1993). Trust Relationships in Secure Systems–A Distributed Authentication Perspective. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy.*

Yahalom R. , Klein B. & Beth T. (1994). Trust-based navigation in distributed systems. *The USENIX Association Computing Systems* **7**(1):45 – 73.

Zar J. H. (1999). *Biostatistical Analysis*. Prentice Hall, forth edn.

Zhou L. & Haas Z. J. (1999). Securing ad hoc networks. *IEEE Networks* .

# A Software design

Here in the software design for the experiment is presented.

## A.1 Input / Output

The input to the experiment program is given in the form of an experiment parameter file. This file is a text (ASCII) file with values for all the parameters. An experiment file could look like this:

```
# Comments can be given by starting a line with #.

NUMBER_OF_NODES = 100
NUMBER_OF_MALICIOUS_NODES = 20
MIN_NUMBER_OF_CONNECTIONS = 10
MAX_NUMBER_OF_CONNECTIONS = 15
NUMBER_TO_EXAMINE = 20
NUMBER_OF_PASSES = 100
CHANCE_OF_EVIDENCE_CREATION =  0.015
CHANCE_OF_EVIDENCE_FALSE = 0.01
EVIDENCE_LIFETIME = 30
REPUTATION_LIFETIME = 15
CONNECTION_FILETIME = 50
REQUEST_TIMEOUT = 2
ACCEPTANCE_LIMIT = 1.5
UNCERTAINTY_LOWER_BOUND = 0
INNER_ACCEPTANCE_LIMIT = 1.5
INNER_UNCERTAINTY_LOWER_BOUND = 0
PARADIGM = EVIDENCE
```

The experiment files are all stored in a directory called *experiment* and have the postfix *.exp*.

Output from the experiment program is given in the form of a series of log files, in text (ASCII) format. The files are created in a directory called *log* and have a name beginning with the experiment filename followed by an underscore ('_')the log name and the postfix *.log*. The log files produced are as follows:

**benign** logs the classifications of good nodes, on each pass. Mainly for debugging purposes-

**malicious** logs the classifications of bad nodes, on each pass. Mainly for debugging purposes.

**min_mean_max** logs minimum, mean and maximum values for the classifications for good and bad nodes. Mainly for debugging purposes.

**percents** logs percent values of classifications for each pass. Mainly for debugging purposes.

**scale** logs the scalability of the experiment as number of calculations and the amount of data sent, as defined in section 6.1

**Sec_Avail** logs security and availability as defined in section 6.2

All log files start with a log header stating which experiment file the log is for as well as the time and date that the experiment was executed.

## A.2 Class diagram

A class diagram of the implementation can be seen in figure 22

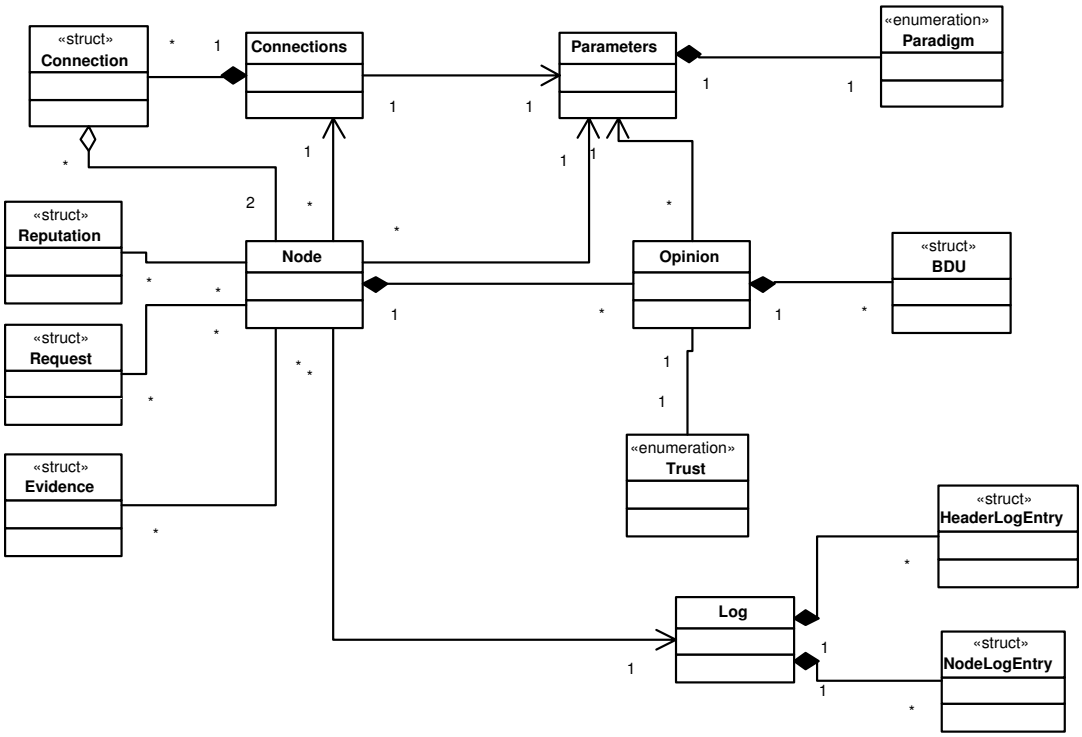


Figure 22: Class diagram of experiment program.

The classes and structures, and their purpose is as follows:

**Connections** creates and stores both the simulated network connections as well as which nodes are to evaluate which other nodes.

**Connection** a structure to represent a connection between two nodes.

**Parameters** reads parameters from an experiment parameter file, and stores the parameter values for the experiment.

**Paradigm** an enumeration for representing the reputation spreading paradigm.

**Node** represents a network node and holds the function for a node to run a pass.

**Reputation** a structure representing a reputation in the form of an opinion.

**Request** a structure representing a request, for either opinions or evidences.

**Evidence** a structure representing an evidence about a nodes behaviour.

**Opinion** represents an opinion. Has functions for calculating an opinion from evidences as well as to calculate the consensus between two opinions. Also has functions for performing an acceptance test.

**BDU** a structure representing an opinion as belief, disbelief and uncertainty.

**Trust** an enumeration representing a classification of an opinion as either accepted, rejected or unclassified.

**Log** a class used to create the log files from the experiment.

**HeaderLogEntry** a log entry for the log header, giving experiment name, date, and time.

**NodeLogEntry** a log entry giving a nodes state at a pass. Used to produce logs, either directly or by summarising them.

For more details on the implementation see the code in appendix B.

# B Code for the experiment

## B.1 Parameters.h

```
#pragma once

//#include <windows.h>

#define MAX_PATH 260

#ifndef _WIN32

#include <strings.h>
inline int stricmp(const char * s1, const char *s2){return
     strcasecmp(s1, s2);}

#endif

#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

enum Paradigm{
    P_Evidence,
    P_Reputation
};

class Parameters
{
public:
    Parameters(void);
    ~Parameters(void);
    // Folder to read experiment files from.
    static char* FOLDER;
    static char* LOG_FOLDER;
    static char FILE_NAME[MAX_PATH];
    static char FILE_BASE[MAX_PATH];

    //Parameters
    static unsigned long NUMBER_OF_NODES;
    static unsigned long NUMBER_OF_MALICIOUS_NODES;
    static unsigned long MIN_NUMBER_OF_CONNECTIONS;
    // The maximum number of connections a node can have.
    static unsigned long MAX_NUMBER_OF_CONNECTIONS;
    // Number of nodes to examine for each node. Including the
         ones they are connected to.
    static unsigned long NUMBER_TO_EXAMINE;
    // The number of passes to run the program for.
    static unsigned long NUMBER_OF_PASSES;
    // The chance that a new piece of evidence is created.
    static double CHANCE_OF_EVIDENCE_CREATION;
    // The chance that a created evidence is false.
    static double CHANCE_OF_EVIDENCE_FALSE;
    // The time for an evidence to live in number of passes.
    static unsigned long EVIDENCE_LIFETIME;
    // Life time for spreaded reputation.
    static unsigned long REPUTATION_LIFETIME;
    // Life time for connections.
    static unsigned long CONNECTION_LIFETIME;
```

```
    // Request timeout as number of hops.
    static unsigned long REQUEST_TIMEOUT;
    // The limit for accepting a node as not benign as a distance
         from {1,0,0).
    static double ACCEPTANCE_LIMIT;
    // The lower limit for when accepting that a node is
         malicious.
    static double UNCERTAINTY_LOWER_BOUND;
    // Inner limits used to not let nodes go to uncertainty
    // before asking for more evidence.
    static double INNER_ACCEPTANCE_LIMIT;
    static double INNER_UNCERTAINTY_LOWER_BOUND;
    // Paradigm to use.
    static Paradigm PARADIGM;
    // Initiate for search of files.
    void init(int argc, char *argv[]);
    // Reads next experiment file, returns NULL if no experiment
         file exists
    bool readFile(void);

private:
    static vector<char *> *m_vFilebase;
};
```

## B.2 node.h

```
#ifndef NODE_H
#define NODE_H

/*
 * Implements a simulated node
 */

#include <vector>
#ifdef _WIN32
#include <hash_set>
#include <hash_map>
using namespace stdext;
#else
#include <ext/hash_set>
#include <ext/hash_map>
using namespace __gnu_cxx;
#endif

#include "opinion.h"

typedef hash_map<unsigned long, vector<Evidence*>*>
     hash_evidence;


struct Request{
    unsigned long nid;
    unsigned long rid;
    unsigned long live;
    hash_set<unsigned long> history;
};

struct Reputation{
    // id of the node that created the reputation
    unsigned long nodeid;
    // Id of the node about which the reputation is an opinion
    unsigned long objectid;
    // The opinion about the node
    BDU reputation;
    // The expiery pass of the reputation.
    unsigned long expiery;
};

class Node
{
    public:
        // class constructor
        Node(unsigned long id, bool malicious);
        // class destructor
        ~Node();

        // Nodes Id.
        unsigned long m_id;
        // Is node malicious.
        bool m_malicious;
        // opinions about other nodes.
        Opinion** m_opinions;
        // Evidence about other nodes.
        // Stored as a hash_map with id as key for fast retreval.
        hash_evidence m_evidence;
```

```
        // create an evidence or return NULL if no evidence is
             created.
        Evidence* createEvidence(unsigned long id, unsigned long
             pass);
        // Request evidence.
        vector<Evidence*> evidenceRequest(Request er);

        Reputation reputationRequest(Request er, unsigned long
             pass);
        // Run a pass.
        void pass(unsigned long pass);

        unsigned long m_evidenceId;
        // remove doulbes from evidence
        void cleanEvidence(unsigned long id);
private:
    unsigned long *m_reputationExpiery;
    void addEvidence(Evidence *e);
    void addEvidence(unsigned long id, vector<Evidence *> e);
};

#endif // NODE_H
```

## B.3 opinion.h

```
#ifndef OPINION_H
#define OPINION_H

#include <iostream>

using namespace std;
/*
 * Opinion space implementation.
 * For Martin Håkanssons' Bachelor thesis.
 */

struct BDU{
        double b;
        double d;
        double u;
    };


struct Evidence{
    unsigned long evidence_id;
    unsigned long node_id; // Id of the node who created the
         evidence.
    bool positive;
    unsigned long object_id; // Id of the node about which the
         evidence prooves something.
    unsigned long expiery; // Expiery time for id.
};

enum Trust{
    T_accepted,
    T_not_accepted,
    T_uncertain
};

class Opinion
{
public:

    // class constructor
    Opinion(unsigned long node_id,unsigned long object);
    Opinion(unsigned long node_id,unsigned long object, double
         b,double d, double u);
    Opinion(unsigned long node_id,unsigned long object, unsigned
         long r, unsigned long s);
    // class destructor
    ~Opinion();

    void evidence2opinion(unsigned long r, unsigned long s);

    // Sets the opinion for the Opinion object.
    void setOpinion(BDU newOpinion);
    // Returns the opinion as a BDU value.
    BDU getOpinion(void);
    // Id of the node about which the opinion is.
    unsigned long m_object;
    // The trust value of the opinion as Accepted, not accepted
         or uncertain.
    Trust accepted(void);
    // Is the opinion near uncertain?
    bool nearUncertain();
```

```
    // Consensus between the opinion and opinion a.
    void consensus(BDU a);

private:
    BDU opinion;


};

#endif // OPINION_H
```

## B.4 connections.h

```
#pragma once

#include <vector>
#ifdef _WIN32
#include <hash_map>
using namespace stdext;
#else
#include <ext/hash_map>
using namespace __gnu_cxx;
#endif

#include "node.h"

// Connections are bidirectional, examinations are not.
struct Connection{
    unsigned long id1;
    unsigned long id2;
};

class Connections
{
public:
    Connections(void);
    ~Connections(void);
    // List of all nodes
    static vector<Node*>* m_nodes;
    // Connections
    static hash_map<unsigned long, vector<int> >* m_connections;
    // Nodes to examine
    static hash_map<unsigned long, vector<int> >* m_examine;
    // Add a new node.
    void newNode(Node* node);
    // Create connections between nodes.
    void createConnections(void);
    // Get a vector of ids connected to node id.
    static vector<int> getConnected(int id);
    // Get a vector of ids to examine, include the connected ids.
    static vector<int> getNodesToExamine(int id);
};
```

## B.5 connections2.h

```
#pragma once

//#include "parameters.h"
#include <vector>

#include "node.h"

class Connections
{
public:
    Connections(void);
    ~Connections(void);
    // List of all nodes ids
    vector<Node> m_nodes;
    // Add a new node.
    void newNode(Node* node);
    // Create connections between nodes.
    void createConnections(void);
};
```

## B.6 Log.h

```
#pragma once

#include <vector>

using namespace std;

// Structures for log entries.
struct HeaderLogEntry{
    char* text;
};

struct NodeLogEntry{
    unsigned long node_id;
    unsigned long pass;
    unsigned long accepted;
    unsigned long accepted_correct;
    unsigned long not_accepted;
    unsigned long not_accepted_correct;
    unsigned long uncertain;
    unsigned long uncertain_good;
    unsigned long benign;
    unsigned long malicious;
};

class Log
{
public:
    Log(void);
    ~Log(void);

    // initialise variables
    static void init(void);
    // Add a node log entry
    static void addLogEntry(NodeLogEntry logEntry);
    // Add a header log entry
    static void addLogEntry(HeaderLogEntry logEntry);
        // Write log to log stream
    static void log(void);
    // Clear log data
    static void destroy(void);
    static void addData(unsigned long i);
    static void addCalc();


private:
    // Logs benign node statistics
    static void logBenign(void);
    // Logs malicious node statistics
    static void logMalicious(void);
        // Node log entries
    static vector <NodeLogEntry>* m_nodeLog;
    // Header log entries
    static vector <HeaderLogEntry>* m_headerLog;
    // write log header to logfile.
    static void logHeader(void);
    // Compile and log node data.
    static void logNodes(void);
    static void logScalability();

    static ofstream m_sLog;
```

```
    static unsigned long calculations;
    static unsigned long dataSent;
    // Logs maximum mean and minimum for wrongly and correctly
         classified nodes
    static void logMaxMeanMin(void);
    static void logSecAvail(void);

};
```

## B.7 experiment.cpp

```
// experiment.cpp : Defines the entry point for the console
     application.
//

#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <time.h>
#include <stdio.h>

#include "Parameters.h"
#include "node.h"
#include "connections.h"
#include "Log.h"

int main(int argc, char* argv[])
{
    time_t timer=0;

    // Initialize parameters..
    Parameters p;
    p.init(argc, argv);
    while(p.readFile()){
        Log::init();

        // Add header log entry for start..
        {
            HeaderLogEntry le;
            le.text=new char[1024];
            strcpy(le.text, "Experiment file: ");
            strcat(le.text, p.FILE_NAME);
            strcat(le.text, " executed: ");

            time_t t = time(NULL);

            tm* lt = localtime(&t);
            char tmp[256];

            sprintf(tmp, "%4d%02d%02d %02d:%02d",
                lt->tm_year+1900, lt->tm_mon+1, lt->tm_mday,
                lt->tm_hour, lt->tm_min);

            strcat(le.text, tmp);

            Log::addLogEntry(le);


            cout << tmp <<": Initializing "<<p.FILE_NAME<<" ..";
            cout.flush();


        }


        // Initialization..

        timer = time(NULL);

        // Initialize pseudorandom generator..
```

```
srand( (unsigned)time( NULL ) );

// Create nodes..

Connections con;

for(unsigned long i=0;
     i<Parameters::NUMBER_OF_MALICIOUS_NODES; i++){
    con.newNode(new Node(i, true));
}
for(unsigned long i =
     Parameters::NUMBER_OF_MALICIOUS_NODES; i<Parameters::NUMBER_OF_NODES; i++){
    con.newNode(new Node(i, false));
}

cout<< "done in "<<(long)(time(NULL)-timer)<<"
     seconds!"<<endl;

cout << "Running../";
cout.flush();

timer=time(NULL);
char c='/'; // Used to show we're alive.

// Run passes and log..
for(unsigned long pass=0;
     pass<Parameters::NUMBER_OF_PASSES; pass++){

    // Create connections if it is time..
    if((pass % Parameters::CONNECTION_LIFETIME)==0){
        con.createConnections();
    }

    // Run pass for every node..
    for(vector<Node*>::iterator i = con.m_nodes->begin();
         i!=con.m_nodes->end(); i++){
        (*i)->pass(pass);

        // Show we're alive..
        cout<<"\b"<<c;
        cout.flush();

        switch(c){
            case '/':
                c='-';
                break;
            case '-':
                c='\\';
                break;
            case '\\':
                c='|';
                break;
            case '|':
                c='/';
                break;
        }

    }
}

cout << "\bdone in " << (long)(time(NULL)-timer) << "
     seconds!" << endl;
```

```
        cout << "Writing log..";
        cout.flush();
        timer=time(NULL);

        // Add header log entry for stop..
        {
            HeaderLogEntry le;

            le.text = new char[1024];
            char tmp[256];

            time_t t = time(NULL);

            tm* lt = localtime(&t);

            sprintf(tmp, "%4d%02d%02d %02d:%02d",
                lt->tm_year+1900, lt->tm_mon, lt->tm_mday,
                lt->tm_hour, lt->tm_min);

            strcpy(le.text, "Finished at: ");
            strcat(le.text, tmp);
            Log::addLogEntry(le);
        }

        // Write log..
        Log::log();

        // Clean up..

        Log::destroy();
        cout << "done in "<<(long)(time(NULL)-timer)<<"
             seconds!"<< endl;
    }


#ifdef _WIN32
    cout<<"\b"<<endl;
    //system("PAUSE");
#else
    cout<<"Finished!\a"<<endl;
#endif
    return 0;
}
```

## B.8 node.cpp

```
#include <stdlib.h>
#include <time.h>
#include <assert.h>

#include "node.h" // class's header file
#include "opinion.h"
#include "Parameters.h"
#include "connections.h"
#include "Log.h"

// class constructor

Node::Node(unsigned long id, bool malicious)
: m_id(id), m_malicious(malicious)
{
    m_opinions = new Opinion*[Parameters::NUMBER_OF_NODES];

    for(unsigned long i = 0; i<Parameters::NUMBER_OF_NODES; i++){
        m_opinions[i]= new Opinion(m_id, i);
    }

    m_evidenceId = 0;

    m_reputationExpiery = new unsigned
         long[Parameters::NUMBER_OF_NODES];
    for(unsigned int i=0; i<Parameters::NUMBER_OF_NODES;
         i++){m_reputationExpiery[i]=0;}

}

// class destructor
Node::~Node()
{
    for(hash_evidence::iterator i=m_evidence.begin();
        i!=m_evidence.end(); i++){
            i->second->clear();

        }

    m_evidence.clear();

    for(unsigned long i = 0; i<Parameters::NUMBER_OF_NODES; i++)
        delete m_opinions[i];

    delete [] m_opinions;

    delete [] m_reputationExpiery;

}


// create an evidence or return NULL if no evidence is created.

Evidence* Node::createEvidence(unsigned long id, unsigned long
     pass)
{

    if(((double)rand()/(double)RAND_MAX)<=Parameters::CHANCE_OF_EVIDENCE_CREATION)
    {
```

```
        Evidence* e=new Evidence;
        e->evidence_id=m_evidenceId++;
        e->node_id=m_id;
        e->object_id=id;
        bool evidence_false =
             ((double)rand()/(double)RAND_MAX)<=Parameters::CHANCE_OF_EVIDENCE_FALSE;

        if(evidence_false)
            e->positive =
                 (*Connections::m_nodes)[id]->m_malicious;
        else
            e->positive =
                 !(*Connections::m_nodes)[id]->m_malicious;

        // Reverse evidence if node malicious.
        if(m_malicious)
            e->positive = !e->positive;

        e->expiery = pass+Parameters::EVIDENCE_LIFETIME;

        return e;
    }
    return NULL;
}


// Run a pass.
void Node::pass(unsigned long pass)
{
    // Do all simulation stuff.

    vector<int> connections=Connections::getConnected(m_id);



    // Try to create evidence about known nodes..
    for(vector<int>::iterator i=connections.begin();
         i!=connections.end(); i++){

        Evidence* ev=createEvidence(*i, pass);

        if(ev!=NULL){
            addEvidence(ev);
        }
    }


    // Create a log entry.
    NodeLogEntry ld;
    ld.node_id=m_id;
    ld.accepted=ld.not_accepted=ld.uncertain=ld.uncertain_good=0;
    ld.accepted_correct=ld.not_accepted_correct=0;
    ld.malicious=ld.benign=0;
    ld.pass=pass;


    // Examine nodes in examine vector..
    vector <int> examine=Connections::getNodesToExamine(m_id);

    Request er;
    Node *m = NULL;
```

```
unsigned long r, s;

for(vector<int>::iterator
     i=examine.begin();i!=examine.end();i++){

    assert(*i!=m_id);


    // If node near uncertain then request further evidence..
    if(m_opinions[*i]->nearUncertain() ||
         (Parameters::PARADIGM==P_Reputation && m_reputationExpiery[*i]>=pass)){

        if(Parameters::PARADIGM == P_Reputation){
            r=0;s=0;

            // Examine and count evidence about node
            hash_evidence::iterator
                 ev_it=m_evidence.find(*i);
            if(ev_it!=m_evidence.end()){
                for(vector<Evidence *>::iterator
                     e=ev_it->second->begin(); e!=ev_it->second->end(); e++){
                    if((*e)->positive)

                        r++;
                    else
                        s++;
                }
            }
            // Calculate own opinion about node.
            m_opinions[*i]->evidence2opinion(r, s);
            m_reputationExpiery[*i]=pass +
                 Parameters::REPUTATION_LIFETIME;


        }

        for(vector<int>::iterator
             j=connections.begin();j!=connections.end();j++){

            if(*j == m_id ||
                *j == *i ||
                m_opinions[*j]->accepted() == T_not_accepted)
                continue;

            m=(*Connections::m_nodes)[*j];

            er.history.insert(m_id);
            er.live=Parameters::REQUEST_TIMEOUT-1;
            er.nid=m_id;
            er.rid=*i;

            if(Parameters::PARADIGM==P_Evidence){
                vector <Evidence *>
                     tmp=m->evidenceRequest(er);
                addEvidence(*i, tmp);
            }
            else if(Parameters::PARADIGM==P_Reputation){
                Reputation tmp = m->reputationRequest(er,
                     pass);

                m_opinions[*i]->consensus(tmp.reputation);
```

```
            }

            er.history.clear();
        }
    }

    if(Parameters::PARADIGM == P_Evidence){
        cleanEvidence(*i);

        r=0;s=0;

        // Examine and count evidence about node
        hash_evidence::iterator ev_it=m_evidence.find(*i);
        if(ev_it!=m_evidence.end()){
            for(vector<Evidence *>::iterator
                 e=ev_it->second->begin(); e!=ev_it->second->end(); e++){
                if((*e)->positive)
                    r++;
                else
                    s++;
            }
        }

        // Calculate opinion about node.
        m_opinions[*i]->evidence2opinion(r, s);
    }


    // Add trust to log entry.
    Node *n=(*Connections::m_nodes)[*i];

    switch(m_opinions[*i]->accepted()){
        case T_uncertain:
                ld.uncertain++;
                if(!(n->m_malicious))
                    ld.uncertain_good++;
                break;
            case T_not_accepted:
                ld.not_accepted++;
                if(n->m_malicious){
                    ld.not_accepted_correct++;
                }
                break;
            case T_accepted:
                ld.accepted++;
                if(!(n->m_malicious)){
                    ld.accepted_correct++;
                }
                break;
    }
    if(n->m_malicious)
        ld.malicious++;
    else
        ld.benign++;
}

// Add pass to log.

if(!m_malicious) // Log only benign nodes.
    Log::addLogEntry(ld);

// Remove old evidences..
```

```
    for(hash_evidence::iterator ev_it=m_evidence.begin();
         ev_it!=m_evidence.end(); ev_it++){
        for(vector<Evidence *>::iterator
             i=ev_it->second->begin();i!=ev_it->second->end();i++){
            if(pass>=(*i)->expiery){
                i=ev_it->second->erase(i);
                i--;
            }
        }
    }
}


vector<Evidence*> Node::evidenceRequest(Request er)
{
    vector<Evidence*> ret(0);

    er.history.insert(m_id);

    if(er.live>0){

        vector<int> connections=Connections::getConnected(m_id);

        er.live--;
        for(unsigned int i=0;i<connections.size();i++){
            Node *n=(*Connections::m_nodes)[connections[i]];


            if((n->m_id == er.rid) ||
                 (m_opinions[n->m_id]->accepted() == T_not_accepted))
                continue;


            if(er.history.find(n->m_id)==er.history.end()){
                vector<Evidence*> tmp=n->evidenceRequest(er);
                addEvidence(er.rid, tmp);
            }
        }
        cleanEvidence(er.rid);
    }

    hash_evidence::iterator i=m_evidence.find(er.rid);
    if(i==m_evidence.end())
        return ret;

    ret.insert(ret.end(), i->second->begin(), i->second->end());

    Log::addData((unsigned long)ret.size());
    return ret;
}

Reputation Node::reputationRequest(Request er,unsigned long pass)
{
    // Calculate own opinion, if needed..

    if(m_reputationExpiery[er.rid]>=pass){
        int r=0, s=0;

        hash_evidence::iterator ev_it=m_evidence.find(er.rid);
        if(ev_it!=m_evidence.end()){
            for(vector<Evidence *>::iterator
                 e=ev_it->second->begin(); e!=ev_it->second->end(); e++){
```

```
                if((*e)->positive)
                    r++;
                else
                    s++;
            }
        }

        m_opinions[er.rid]->evidence2opinion(r, s);
        m_reputationExpiery[er.rid] = pass +
             Parameters::REPUTATION_LIFETIME;
    }

    Reputation ret;
    ret.nodeid = m_id;
    ret.objectid = er.rid;

    er.history.insert(m_id);

    if(er.live>0){

      vector<int> connections=Connections::getConnected(m_id);


        er.live--;
        for(unsigned int i=0;i<connections.size();i++){
            Node *n=(*Connections::m_nodes)[connections[i]];

            if(m_opinions[n->m_id]->accepted()==T_not_accepted)
                continue;

            if(er.history.find(n->m_id)==er.history.end()){
                Reputation tmp=n->reputationRequest(er, pass);

                m_opinions[er.rid]->consensus(tmp.reputation);

            }
        }
    }


    //BDU rep = op->getOpinion();

    ret.reputation = m_opinions[er.rid]->getOpinion();




    Log::addData(1);
    return ret;
}

// remove doubles from evidence
inline void Node::cleanEvidence(unsigned long id)
{
    hash_evidence::iterator it=m_evidence.find(id);

    if(it==m_evidence.end())
        return;

    for(vector<Evidence *>::iterator
         i=it->second->begin();i!=it->second->end();i++){
        for(vector<Evidence *>::iterator
```

```
             j=i+1;j!=it->second->end();j++){
            if(((*i)->node_id == (*j)->node_id) &&
                 ((*i)->evidence_id == (*j)->evidence_id)){
                j=it->second->erase(j);
                j--;
            }
        }
    }
}

void Node::addEvidence(Evidence *e){
    hash_evidence::iterator i = m_evidence.find(e->object_id);

    if(i==m_evidence.end()){
        i =
            m_evidence.insert(hash_evidence::value_type(e->object_id, new vector<Evidence *>(0))).first;
    }
    i->second->push_back(e);
}

void Node::addEvidence(unsigned long id, vector <Evidence *> e){
    hash_evidence::iterator i=m_evidence.find(id);

    if(i==m_evidence.end()){
        i=m_evidence.insert(hash_evidence::value_type(id, new
             vector<Evidence *>(0))).first;
    }
    i->second->insert(i->second->end(), e.begin(), e.end());
}
```

## B.9 opinion.cpp

```
#include <math.h>
#include "opinion.h" // class's header file
#include "Parameters.h"
#include "Log.h"



// class constructors
Opinion::Opinion(unsigned long node_id,unsigned long object)
: m_object(object)
{
    // Create a dummy trust value..
    opinion.b=0;
    opinion.d=0;
    opinion.u=1;

}

Opinion::Opinion(unsigned long node_id,unsigned long object,
     double b, double d, double u)
: m_object(object)
{
    opinion.b=b;
    opinion.d=d;
    opinion.u=u;
}

Opinion::Opinion(unsigned long node_id,unsigned long object,
     unsigned long r, unsigned long s)
: m_object(object)
{
    evidence2opinion(r, s);

}

// class destructor
Opinion::~Opinion()
{
    // insert your code here

}


void Opinion::evidence2opinion(unsigned long r, unsigned long s)
{
    // r are positive evidences, s are negative evidencesö.
    Log::addCalc();

    opinion.b = (double)r /(double)(r+s+1.0);
    opinion.d = (double)s / (double)(r+s+1.0);
    opinion.u = 1.0 /(double)(r+s+1.0);
}



// Sets the opinion for the Opinion object.
void Opinion::setOpinion(BDU newOpinion)
{
    opinion.b=newOpinion.b;
    opinion.d=newOpinion.d;
```

```
    opinion.u=newOpinion.u;
}


BDU Opinion::getOpinion(void)
{
    BDU ret;
    ret.b=opinion.b;
    ret.d=opinion.d;
    ret.u=opinion.u;

    return ret;
}

Trust Opinion::accepted(void)
{
    Log::addCalc();
    // Check acceptance by checking distance from point {1.0,
         0.0, 0.0}.
    double db = opinion.b - 1.0;
    double dd = opinion.d;
    double du = opinion.u;

    if(sqrt((db * db)+(dd * dd)+(du * du)) <=
         Parameters::ACCEPTANCE_LIMIT)
        return T_accepted; // Accepted
    else if(opinion.u <= Parameters::UNCERTAINTY_LOWER_BOUND)
        return T_not_accepted;    // Not accepted

    return T_uncertain; // Uncertain
}

bool Opinion::nearUncertain()
{
    Log::addCalc();
    double db = opinion.b - 1.0;
    double dd = opinion.d;
    double du = opinion.u;

    if(sqrt((db * db)+(dd * dd)+(du * du)) <=
         Parameters::INNER_ACCEPTANCE_LIMIT)
        return false;
    if(opinion.u<=Parameters::INNER_UNCERTAINTY_LOWER_BOUND)
        return false;

    return true; // Near uncertain
}

void Opinion::consensus(BDU a)
{
    Log::addCalc();
    double K = a.u + opinion.u - a.u * opinion.u;
    if(K==0.0){
        // Means that both opinions are "blind" and
        // therefor a consensus can not be reached.
        return;
    }

    opinion.b = (a.b * opinion.u + opinion.b * a.u) / K;
    opinion.d = (a.d * opinion.u + opinion.d * a.u) / K;
    opinion.u = (a.u * opinion.u) / K;
```

```
}
```

## B.10 connections.cpp

```
#include <algorithm>
#include "connections.h"
#include "Parameters.h"

vector<Node*>* Connections::m_nodes=NULL;
hash_map<unsigned long, vector<int> >*
     Connections::m_connections=NULL;
hash_map<unsigned long, vector<int> >*
     Connections::m_examine=NULL;

Connections::Connections(void)
{
    m_nodes=new vector<Node*>(0);
    m_connections=new hash_map<unsigned long, vector<int> >;
    m_examine = new hash_map<unsigned long, vector<int> >;
}

Connections::~Connections(void)
{
    while(!m_nodes->empty()){
        delete (*m_nodes)[0];
        m_nodes->erase(m_nodes->begin());
    }
    delete m_nodes;
    m_connections->clear();
    delete m_connections;

    m_examine->clear();
    delete m_examine;
}

// Add a new node.
void Connections::newNode(Node* node)
{
    m_nodes->push_back(node);
}

// Create connections between nodes.
void Connections::createConnections(void)
{
    unsigned long number_of_nodes = (unsigned
         long)m_nodes->size();

    // Create the connections between the nodes.
    int* m_number_of_connections = new int[number_of_nodes];

    for(unsigned long i=0;i<number_of_nodes;i++)
         m_number_of_connections[i]=0;

    //Create a shuffled node vector.

    vector<Connection> connections(0);
    vector<Connection> examine(0);

    vector<Node *> shuffled(0);
    for(unsigned int i=0; i<m_nodes->size(); i++)
         shuffled.push_back((*m_nodes)[i]);

    random_shuffle(shuffled.begin(), shuffled.end());
```

```
for(unsigned long i=0;i<number_of_nodes;i++){
    unsigned long n;
    if(Parameters::MIN_NUMBER_OF_CONNECTIONS ==
         Parameters::MAX_NUMBER_OF_CONNECTIONS)
        n=Parameters::MIN_NUMBER_OF_CONNECTIONS;
    else
        n=((rand() % (Parameters::MAX_NUMBER_OF_CONNECTIONS -
             Parameters::MIN_NUMBER_OF_CONNECTIONS)) + Parameters::MIN_NUMBER_OF_CONNECTIONS);

    for(unsigned long j = m_number_of_connections[i];
         j<Parameters::NUMBER_TO_EXAMINE; j++){

        unsigned long to=i + j + 1;

        do{
            if(to == i) to++;
            if(to >= number_of_nodes){
                to = to - number_of_nodes;
            }
        }while(to==i || to>=number_of_nodes);


        Connection c = {shuffled[i]->m_id,
             shuffled[to]->m_id};

        if(j < n){
            connections.push_back(c);
            m_number_of_connections[to]++;
            m_number_of_connections[i]++;
        }else{
            m_number_of_connections[i]++;
            examine.push_back(c);
        }
    }

}

// Erase doubles.
for(unsigned int i=0; i<connections.size();i++){
    for(unsigned int j=i+1; j<connections.size(); j++){
        if((connections[i].id1==connections[j].id1 &&
             connections[i].id2 == connections[j].id2) ||
            (connections[i].id1 == connections[j].id2 &&
                 connections[i].id2 == connections[j].id1)){
                connections.erase(connections.begin() + j);
                j--;
            }
    }
}

// Create hash maps.

for(unsigned long i = 0; i<number_of_nodes;i++){
    vector<int> con(0);

    for(unsigned int j=0;j<connections.size();j++){
        if(connections[j].id1==i)
            con.push_back(connections[j].id2);
        else if((connections)[j].id2==i)
            con.push_back(connections[j].id1);
    }
```

```
        vector<int> ex(0);
        ex.insert(ex.end(), con.begin(), con.end());

        // Add other nodes to examine.
        for(unsigned int j=0;j<examine.size();j++){
            if((examine)[j].id1 == i)
                ex.push_back(examine[j].id2);
        }

        m_connections->insert(hash_map<unsigned long, vector<int>
             >::value_type(i, con));
        m_examine->insert(hash_map<unsigned long, vector<int>
             >::value_type(i, ex));
    }
}


// Get a vector of ids connected to node id.
vector<int> Connections::getConnected(int id)
{
    return m_connections->find(id)->second;
}

vector<int> Connections::getNodesToExamine(int id)
{
    return m_examine->find(id)->second;
}
```

## B.11 Parameters.cpp

```
#include "Parameters.h"
#include <string.h>

vector <char *> *Parameters::m_vFilebase;

char* Parameters::FOLDER = "experiment/";
char Parameters::FILE_NAME[MAX_PATH];
char Parameters::FILE_BASE[MAX_PATH];
char* Parameters::LOG_FOLDER = "log/";

// Default values.
unsigned long Parameters::NUMBER_OF_NODES = 10;
unsigned long Parameters::NUMBER_OF_MALICIOUS_NODES = 2;
unsigned long Parameters::MIN_NUMBER_OF_CONNECTIONS = 9;
unsigned long Parameters::MAX_NUMBER_OF_CONNECTIONS = 9;
unsigned long Parameters::NUMBER_TO_EXAMINE = 10;
unsigned long Parameters::NUMBER_OF_PASSES = 100;

double Parameters::CHANCE_OF_EVIDENCE_CREATION = 1.0;
double Parameters::CHANCE_OF_EVIDENCE_FALSE = 0.0;
unsigned long Parameters::EVIDENCE_LIFETIME = 1000;
unsigned long Parameters::REPUTATION_LIFETIME = 100;
unsigned long Parameters::CONNECTION_LIFETIME = 100;

unsigned long Parameters::REQUEST_TIMEOUT = 2;
double Parameters::ACCEPTANCE_LIMIT = 0.5;
double Parameters::UNCERTAINTY_LOWER_BOUND = 0.25;
double Parameters::INNER_ACCEPTANCE_LIMIT = 0.5;
double Parameters::INNER_UNCERTAINTY_LOWER_BOUND = 0.25;

Paradigm Parameters::PARADIGM = P_Evidence;


Parameters::Parameters(void)
{
}

Parameters::~Parameters(void)
{
}

void Parameters::init(int argc, char *argv[])
{
    m_vFilebase = new vector<char *>(0);

    for(int i=argc-1;i>0;i--){
        if(stricmp(argv[i], "-f")==0){
            strcpy(FOLDER, argv[i+1]);
            m_vFilebase->pop_back();
        }
        m_vFilebase->push_back(argv[i]);
    }
}


// Reads next experiment file, returns NULL if no experiment file
     exists
bool Parameters::readFile(void)
{
```

```
// Some inits first..

if(m_vFilebase->empty())
    return false;

strcpy(FILE_BASE, (char *)m_vFilebase->back());
m_vFilebase->pop_back();

strcpy(FILE_NAME, FOLDER);
strcat(FILE_NAME, FILE_BASE);
strcat(FILE_NAME, ".exp");
ifstream config(FILE_NAME);

// We got a file read parameters from it.


if(!config || !config.is_open())
    return false;


char line[256];
char* pLine;
char parameter[256];
char* value;

for(config.getline(line, 255, '\n');
    !config.eof();
    config.getline(line, 255, '\n')){

    pLine=line;
    while(line[0]==' ' || line[0]=='\t') pLine++; // Removes
         whitespace

    if(pLine[0]=='#' || strcmp(pLine, "")==0) // Comment or
         empty line.
        continue;

    value=strchr(pLine, '=');
    strncpy(parameter, pLine, value-pLine);
    parameter[(value-pLine)]='\0';
    for(size_t i=strlen(parameter)-1;parameter[i]==' ' ||
        parameter[i]=='\t';parameter[i--]='\0'); // remove
             trailing whitespace
    value++;

    if(stricmp(parameter, "NUMBER_OF_NODES")==0)
        NUMBER_OF_NODES=atoi(value);
    else if(stricmp(parameter,
         "NUMBER_OF_MALICIOUS_NODES")==0)
        NUMBER_OF_MALICIOUS_NODES=atoi(value);
    else if(stricmp(parameter,
         "MIN_NUMBER_OF_CONNECTIONS")==0)
        MIN_NUMBER_OF_CONNECTIONS=atoi(value);
    else if(stricmp(parameter,
         "MAX_NUMBER_OF_CONNECTIONS")==0)
        MAX_NUMBER_OF_CONNECTIONS=atoi(value);
    else if(stricmp(parameter, "NUMBER_TO_EXAMINE")==0)
        NUMBER_TO_EXAMINE= atoi(value);
    else if(stricmp(parameter, "NUMBER_OF_PASSES")==0)
        NUMBER_OF_PASSES=atoi(value);
```

```
        else if(stricmp(parameter,
             "CHANCE_OF_EVIDENCE_CREATION")==0)
            CHANCE_OF_EVIDENCE_CREATION=atof(value);
        else if(stricmp(parameter,
             "CHANCE_OF_EVIDENCE_FALSE")==0)
            CHANCE_OF_EVIDENCE_FALSE = atof(value);
        else if(stricmp(parameter, "EVIDENCE_LIFETIME")==0)
            EVIDENCE_LIFETIME=atoi(value);
        else if(stricmp(parameter, "REPUTATION_LIFETIME")==0)
            REPUTATION_LIFETIME=atoi(value);
        else if(stricmp(parameter, "CONNECTION_LIFETIME")==0)
            CONNECTION_LIFETIME=atoi(value);

        else if(stricmp(parameter, "REQUEST_TIMEOUT")==0)
            REQUEST_TIMEOUT=atoi(value);
        else if(stricmp(parameter, "ACCEPTANCE_LIMIT")==0)
            ACCEPTANCE_LIMIT=atof(value);
        else if(stricmp(parameter, "UNCERTAINTY_LOWER_BOUND")==0)
            UNCERTAINTY_LOWER_BOUND=atof(value);
        else if(stricmp(parameter, "INNER_ACCEPTANCE_LIMIT")==0)
            INNER_ACCEPTANCE_LIMIT=atof(value);
        else if(stricmp(parameter,
             "INNER_UNCERTAINTY_LOWER_BOUND")==0)
            INNER_UNCERTAINTY_LOWER_BOUND=atof(value);
        else if(stricmp(parameter, "PARADIGM")==0){
            if(strstr(value, "EVIDENCE")!=NULL)
                PARADIGM=P_Evidence;
            if(strstr(value, "REPUTATION")!=NULL)
                PARADIGM=P_Reputation;
        }
        //else if(stricmp(parameter, "")==0);


    }


    config.close();

    return true;
}
```

## B.12 Log.cpp

```
#include <iostream>
#include <float.h>

#ifdef _WIN32
#define isnan(x) _isnan(x)
#endif

#include "Log.h"
#include "Parameters.h"


using namespace std;


// Initialise static variables..
vector <NodeLogEntry>* Log::m_nodeLog=NULL;
vector <HeaderLogEntry>* Log::m_headerLog=NULL;

ofstream Log::m_sLog;
unsigned long Log::calculations=0;
unsigned long Log::dataSent=0;

Log::Log(void)
{
}

Log::~Log(void)
{
}

// initialise variables
void Log::init(void)
{
    if(m_nodeLog != NULL){
        m_nodeLog->clear();
        delete m_nodeLog;
    }
    if(m_headerLog != NULL){
        m_headerLog->clear();
        delete m_headerLog;
    }
    m_nodeLog = new vector <NodeLogEntry>(0);
    m_headerLog = new vector <HeaderLogEntry>(0);
    calculations=0;
    dataSent=0;
}

// Add a node log entry
void Log::addLogEntry(NodeLogEntry logEntry)
{
    m_nodeLog->push_back(logEntry);
}

// Add a header log entry
void Log::addLogEntry(HeaderLogEntry logEntry)
{
    m_headerLog->push_back(logEntry);
}

// write log header to logfile.
```

```
void Log::logHeader(void)
{
    for(unsigned int i = 0; i<m_headerLog->size(); i++){
        m_sLog<<(*m_headerLog)[i].text<<endl;
    }
}

// Compile and log node data.
void Log::logNodes(void)
{

    m_sLog<<"Pass;";
    m_sLog<<"Accepted (mean);Correctly Accepted;";
    m_sLog<<"Not Accepted (mean);Correctly Not accepted;";
    m_sLog<<"Uncertain (mean);Percent uncertain"<<endl;

    for(unsigned long pass = 0;
         pass<Parameters::NUMBER_OF_PASSES; pass++){
        m_sLog<<pass<<";";

        unsigned long accepted=0;
        unsigned long accepted_correct=0;
        unsigned long not_accepted=0;
        unsigned long not_accepted_correct=0;
        unsigned long uncertain=0;

        for(unsigned int i=0; i<m_nodeLog->size();i++){
            NodeLogEntry le=(*m_nodeLog)[i];
            if(le.pass==pass){
                accepted            += le.accepted;
                accepted_correct    += le.accepted_correct;
                not_accepted        += le.not_accepted;
                not_accepted_correct+= le.not_accepted_correct;
                uncertain            += le.uncertain;
            }else if(le.pass>pass)
                break;
        }
        double p_accepted_correct = (double)accepted_correct /
             (double)accepted;

        double p_not_accepted_correct =
            (double)not_accepted_correct / (double)not_accepted;

        double p_uncertain =
            (double)(uncertain) / (double)(accepted +
                 not_accepted + uncertain);

        double mean_accepted =
            (double)(accepted) / (double)
                 (Parameters::NUMBER_OF_NODES);

        double mean_not_accepted =
            (double) (not_accepted) / (double)
                 (Parameters::NUMBER_OF_NODES);

        double mean_uncertain =
            (double) (uncertain) / (double)
                 (Parameters::NUMBER_OF_NODES);

        if(isnan(p_accepted_correct)) p_accepted_correct = 1.0;
        if(isnan(p_not_accepted_correct)) p_not_accepted_correct
             = 1.0;
```

```
        if(isnan(p_uncertain)) p_uncertain = 1.0;

        m_sLog.imbue(locale("Swedish_Sweden"));
        m_sLog<<mean_accepted<<";";
        m_sLog<<p_accepted_correct<<";";
        m_sLog<<mean_not_accepted<<";";
        m_sLog<<p_not_accepted_correct<<";";
        m_sLog<<mean_uncertain<<";";
        m_sLog<<p_uncertain<<endl;

    }
}

// Write log to log stream
void Log::log(void)
{
    char logFile[MAX_PATH];
    strcpy(logFile, Parameters::LOG_FOLDER);
    strcat(logFile, Parameters::FILE_BASE);
    strcat(logFile, "_percents.log");

    m_sLog.open(logFile, ios_base::out | ios_base::trunc);
    logHeader();
    logNodes();

    m_sLog.close();

    strcpy(logFile, Parameters::LOG_FOLDER);
    strcat(logFile, Parameters::FILE_BASE);
    strcat(logFile, "_benign.log");
    m_sLog.open(logFile, ios_base::out | ios_base::trunc);
    logHeader();
    logBenign();
    m_sLog.close();

    strcpy(logFile, Parameters::LOG_FOLDER);
    strcat(logFile, Parameters::FILE_BASE);
    strcat(logFile, "_malicious.log");
    m_sLog.open(logFile, ios_base::out | ios_base::trunc);
    logHeader();
    logMalicious();
    m_sLog.close();

    strcpy(logFile, Parameters::LOG_FOLDER);
    strcat(logFile, Parameters::FILE_BASE);
    strcat(logFile, "_scale.log");
    m_sLog.open(logFile, ios_base::out | ios_base::trunc);
    logHeader();
    logScalability();
    m_sLog.close();

    strcpy(logFile, Parameters::LOG_FOLDER);
    strcat(logFile, Parameters::FILE_BASE);
    strcat(logFile, "_min_mean_max.log");
    m_sLog.open(logFile, ios_base::out | ios_base::trunc);
    logHeader();
    logMaxMeanMin();
    m_sLog.close();


    strcpy(logFile, Parameters::LOG_FOLDER);
    strcat(logFile, Parameters::FILE_BASE);
```

```
    strcat(logFile, "_Sec_Avail.log");
    m_sLog.open(logFile, ios_base::out | ios_base::trunc);
    logHeader();
    logSecAvail();
    m_sLog.close();
}

// Clear log data
void Log::destroy(void)
{
    m_headerLog->clear();
    m_headerLog=NULL;
    m_nodeLog->clear();
    m_nodeLog=NULL;
}

// Logs benign node statistics
void Log::logBenign(void)
{
    m_sLog<<"Pass;Accepted;Not Accepted;Uncertain;Total"<<endl;
    unsigned long total=0;
    unsigned long accepted=0;
    unsigned long not_accepted=0;
    unsigned long uncertain=0;

    for(unsigned long pass = 0; pass<
         Parameters::NUMBER_OF_PASSES; pass++){
        total=accepted=not_accepted=uncertain=0;
        for(unsigned int i=0;i<m_nodeLog->size();i++){
            NodeLogEntry ne = (*m_nodeLog)[i];
            if(ne.pass!=pass)
                continue;

            accepted+=ne.accepted_correct;
            not_accepted+=(ne.not_accepted -
                 ne.not_accepted_correct);
            uncertain+=(ne.uncertain_good);
            total += ne.accepted_correct +
                 (ne.not_accepted-ne.not_accepted_correct) + ne.uncertain_good;
        }

        m_sLog<<pass<<";";
        m_sLog<<accepted<<";";
        m_sLog<<not_accepted<<";";
        m_sLog<<uncertain<<";";
        m_sLog<<total<<endl;
    }
}

// Logs malicious node statistics
void Log::logMalicious(void)
{
    m_sLog<<"Pass;Accepted;Not Accepted;Uncertain;Total"<<endl;
    unsigned long total=0;
    unsigned long accepted=0;
    unsigned long not_accepted=0;
    unsigned long uncertain=0;

    for(unsigned long pass = 0; pass<
         Parameters::NUMBER_OF_PASSES; pass++){
        total=accepted=not_accepted=uncertain=0;
        for(unsigned int i=0;i<m_nodeLog->size();i++){
```

```
            NodeLogEntry ne = (*m_nodeLog)[i];
            if(ne.pass!=pass)
                continue;

            accepted+=(ne.accepted - ne.accepted_correct);
            not_accepted+=ne.not_accepted_correct;
            uncertain+=(ne.uncertain - ne.uncertain_good);

            total += (ne.accepted-ne.accepted_correct) +
                 ne.not_accepted_correct + (ne.uncertain-ne.uncertain_good);
        }

        m_sLog<<pass<<";";
        m_sLog<<accepted<<";";
        m_sLog<<not_accepted<<";";
        m_sLog<<uncertain<<";";
        m_sLog<<total<<endl;
    }
}

void Log::addCalc()
{
    calculations++;
}

void Log::addData(unsigned long i)
{
    dataSent+=i;
}

void Log::logScalability()
{
    m_sLog<<"Calculations; "<<calculations<<endl;
    m_sLog<<"data sent; "<<dataSent<<endl;
}

// Logs maximum mean and minimum for wrongly and correctly
     classified nodes
void Log::logMaxMeanMin(void)
{
    long double maxWronglyAcceptedNodes=0.0;
    long double
         minWronglyAcceptedNodes=(double)Parameters::NUMBER_OF_NODES;
    long double meanWronglyAcceptedNodes=0.0;

    long double maxWronglyNotAcceptedNodes=0.0;
    long double minWronglyNotAcceptedNodes=
         (double)Parameters::NUMBER_OF_NODES;
    long double meanWronglyNotAcceptedNodes=0.0;

    long double meanUncertainNodes=0.0;
    long double minUncertainNodes =
         (double)Parameters::NUMBER_OF_NODES;

    long double tmpWronglyAcceptedNodes=0.0;
    long double tmpWronglyNotAcceptedNodes=0.0;
    long double tmpUncertainNodes=0.0;
    long double tmpNumberOfNodes = 0.0;

    long double tmpTotalWronglyAccpetedNodes=0.0;
    long double tmpTotalWronglyNotAcceptedNodes=0.0;
    long double tmpTotalUncertainNodes=0.0;
```

```
long double tmpTotalNumberOfNodes = 0.0;


for(unsigned long pass= 0; pass <
     Parameters::NUMBER_OF_PASSES; pass++){
    tmpWronglyAcceptedNodes=0.0;
    tmpWronglyNotAcceptedNodes=0.0;
    tmpUncertainNodes = 0.0;
    tmpNumberOfNodes = 0.0;

    for(unsigned int i=0;i<m_nodeLog->size();i++){
        NodeLogEntry ne = (*m_nodeLog)[i];
        if(ne.pass!=pass)
            continue;

        tmpWronglyAcceptedNodes+=(ne.accepted -
             ne.accepted_correct);


        tmpWronglyNotAcceptedNodes+=(ne.not_accepted -
             ne.not_accepted_correct);


        tmpUncertainNodes += ne.uncertain;

        tmpNumberOfNodes += ne.benign + ne.malicious;
    }

    tmpTotalNumberOfNodes += tmpNumberOfNodes;
    tmpTotalWronglyNotAcceptedNodes +=
         tmpWronglyNotAcceptedNodes;
    tmpTotalWronglyAccpetedNodes += tmpWronglyAcceptedNodes;
    tmpTotalUncertainNodes += tmpUncertainNodes;

    tmpWronglyAcceptedNodes = tmpWronglyAcceptedNodes /
         tmpNumberOfNodes;
    tmpWronglyNotAcceptedNodes = tmpWronglyNotAcceptedNodes /
         tmpNumberOfNodes;
    tmpUncertainNodes = tmpUncertainNodes / tmpNumberOfNodes;

    if(tmpWronglyAcceptedNodes > maxWronglyAcceptedNodes)
        maxWronglyAcceptedNodes = tmpWronglyAcceptedNodes;
    if(tmpWronglyAcceptedNodes < minWronglyAcceptedNodes)
        minWronglyAcceptedNodes = tmpWronglyAcceptedNodes;

    if(tmpWronglyNotAcceptedNodes >
         maxWronglyNotAcceptedNodes)
        maxWronglyNotAcceptedNodes =
             tmpWronglyNotAcceptedNodes;
    if(tmpWronglyNotAcceptedNodes <
         minWronglyNotAcceptedNodes)
        minWronglyNotAcceptedNodes =
             tmpWronglyNotAcceptedNodes;

    if(tmpUncertainNodes < minUncertainNodes)
        minUncertainNodes = tmpUncertainNodes;
}

meanWronglyAcceptedNodes =
     (tmpTotalWronglyAccpetedNodes/Parameters::NUMBER_OF_PASSES) / tmpTotalNumberOfNodes;
meanWronglyNotAcceptedNodes =
     (tmpTotalWronglyNotAcceptedNodes / Parameters::NUMBER_OF_PASSES) / tmpTotalNumberOfNodes;
```

```
    meanUncertainNodes = ( tmpTotalUncertainNodes /
         Parameters::NUMBER_OF_PASSES) / tmpTotalNumberOfNodes;



    m_sLog << "Max Accepted malicious nodes; " <<
         maxWronglyAcceptedNodes *100.0<<endl;
    m_sLog << "Min Accepted malicious nodes; " <<
         minWronglyAcceptedNodes*100.0<<endl;
    m_sLog << "Mean Accepted malicious nodes; " <<
         meanWronglyAcceptedNodes*100.0<<endl;

    m_sLog << "Max Not accepted  benign nodes; " <<
         maxWronglyNotAcceptedNodes*100.0<<endl;
    m_sLog << "Min Not accepted benign nodes; " <<
         minWronglyNotAcceptedNodes*100.0<<endl;
    m_sLog << "Mean Not accepted benign nodes; " <<
         meanWronglyNotAcceptedNodes*100.0<<endl;

    m_sLog << "Min uncertain nodes; " <<
         minUncertainNodes*100.0<<endl;
    m_sLog << "Mean uncertain nodes; " <<
         meanUncertainNodes*100.0<<endl;
}

void Log::logSecAvail(void)
{

    unsigned long totalNumberOfBenignNodes = 0;
    unsigned long totalNumberOfMaliciousNodes = 0;

    unsigned long acceptedBenign = 0;
    unsigned long notAcceptedAndUncertainMalicious = 0;


    for(unsigned int i=0;i<m_nodeLog->size();i++){
        NodeLogEntry ne = (*m_nodeLog)[i];

        totalNumberOfBenignNodes += ne.benign;

        acceptedBenign += ne.accepted_correct;

        totalNumberOfMaliciousNodes += ne.malicious;

        notAcceptedAndUncertainMalicious +=
            (ne.malicious - (ne.accepted - ne.accepted_correct));

    }


    long double availability = ((long double)acceptedBenign) /
         ((long double)totalNumberOfBenignNodes);
    long double security = ((long
         double)notAcceptedAndUncertainMalicious) /
        ((long double) totalNumberOfMaliciousNodes);

    m_sLog << "Availability: " << availability * 100.0 << endl;
    m_sLog << "Security: " << security * 100.0 << endl;

}
```