Forward

This thesis presents the result of my master's project entitled *Dynamic feed-back mechanisms in Trust-Based DSR*, which is a five months project from Jan 31st to July 15th 2005 and corresponds to 30 ETCS points. Before that I have been studying the international master's program in Computer System Engineering at DTU and this is my final project.

I would like to appreciate Christian Damsgaard Jensen, who suggested me to work on this interesting project, supervised me in the thesis work and gave me a lot of inspirations.

I would also like to thank Lennart Conrad and Sonja Buchegger, who took time to reply my questions and gave me precious suggestions.

Last but not least, I would like to thank my husband Ye Zhang, who takes over a lot of housework so that I can concentrate on my thesis.

Shanshan Song
Lyngby, DTU, Room 105, July 15th, 2005

Signature: _________________________

Date: _________________________

Abstract

Mobile Ad Hoc Network (MANET) is a collection of wireless mobile nodes that dynamically function as a network without the use of any existing infrastructure and centralized administration. The mobile nodes must cooperate at the routing level in order to forward packets to from source to the destination. Current ad hoc routing protocols such as DSR assumes the network is benign and cannot cope with misbehavior, i.e., a misbehavior node may drop packet silently to save battery power, etc.

Dynamic feedback mechanism has been recently introduced to mitigate the misbehavior in MANET. The idea is to build the trust relationship between the mobile nodes in the MANET and select routes based on the formed trust values. CONFIDANT is a dynamic feedback mechanism in which mobile nodes monitor the behavior of their neighbors and exchange first hand information about other nodes in the MANET. This allows other nodes to change their reputation value accordingly and thus identify the misbehaved nodes.

This thesis investigates dynamic feedback mechanisms as security solutions for MANET, implements CONFIDANT protocol using ns2 as simulation environment, and evaluates the performance of CONFIDANT fortified DSR in the MANET where misbehaved nodes present.

Table of Contents

1 Introduction

1.1 What is Mobile Ad Hoc Network?

With rapid development of wireless technology, the Mobile Ad Hoc Network (MANET) has emerged as a new type of wireless network. MANET is a collection of wireless mobile nodes (e.g. laptops) that dynamically function as a network without the use of any existing infrastructure and centralized administration. It is an autonomous system where each node operates not only as an end system but also as a router to forward packets for other nodes.

Since the nodes in MANET move around, the wireless links break and re-establish frequently. Furthermore, most of mobile nodes are resource limited in computing capability and battery power and therefore traditional computing content routing protocols are not suitable for MANET. Several ad hoc routing protocols have been proposed for each node acting as router and maintaining routing information.

Figure 1-1 shows an example of using MANET to hold conference meeting in a company. A group of mobile device users set up a meeting outside their normal office environment where the business network infrastructure is missing. The mobile devices automatically construct a mobile ad hoc network through wireless links and communicate with one another. The figure shows topology of the network and the available wireless links at a certain time. Suppose Susan wants to send data to Jerry. According to the network topology, Jerry's PDA is not in the immediate radio transmission range of Susan's laptop. The routing software on Susan's laptop finds a route Susan → Tommy → Jerry and sends the data packets to Tommy's laptop. Then Tommy's laptop forwards the packets to the destination, Jerry's PDA. If the network topology changes and the wireless link between Susan and Tommy becomes broken, the routing software on Susan's laptop will try to find anther route, e.g. Susan→ Mary → Jerry.

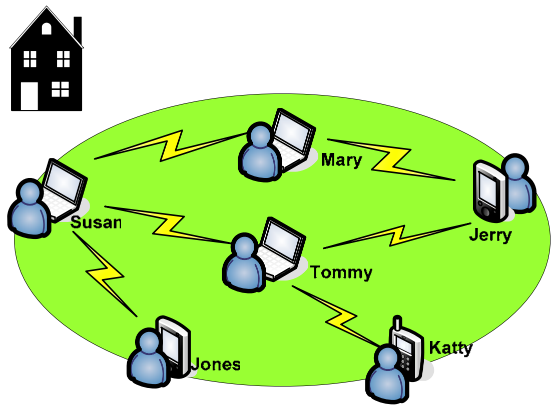


Figure 1-1 Mobile Ad Hoc Network is used in conferencing

There are many other applications of MANET. For examples, MANET can be used to provide ***emergency services*** when the network is impaired due to the damaging of existing infrastructure [8]. Computer scientists have predicted a world of ***ubiquitous computing*** in which computers will be all around us, constantly performing mundane tasks to make our lives a little easier. These ubiquitous computers connect in mobile ad hoc mode and change the environment or react to the change of the environment where they are suited. MANET is also found useful in the so-called ***sensor dust network*** to coordinate the activities and reports of a large collection of tiny sensor devices which could offer detailed information about terrain or environmental dangerous conditions.

1.2 Problem Statement and Motivation

Most current ad hoc routing protocols assume that the wireless network is benign and every node in the network strictly follows the routing behavior and is willing to forward packets for other nodes. Most of these protocols cope well with the dynamically changing topology. However, they do not address the problems when misbehavior nodes present in the network.

A commonly observed misbehavior is packet dropping. In a practical MANET, most devices have very limited computing and battery power while packet forwarding consumes a lot of such resources. Thus some of the mobile devices would not like to forward the packets for the benefit of others and they drop packets not destined to them. On the other hand, they still make use of other nodes to forward packets that they originate. These misbehaved nodes are very difficult to identify because we cannot tell that whether the packets are dropped intentionally by the misbehaved nodes or dropped due to the node having moved out of transmission range or other link error. Packet drop significantly decreases the network performance.

Traditional security mechanisms are generally not suitable for MANET because:

1) The network lacks central infrastructure to apply traditional security mechanism such as access control, authentication and trusted third party.
2) Limited bandwidth, battery lifetime, and computation power prohibits the deployment of complex routing protocols or encryption algorithms. New security models or mechanisms suitable for MANET must be found.
3) Network topologies and memberships are constantly changing. Thus new intrusion detection system and entity recognition mechanisms that are suitable for mobile ad hoc networks must be designed to avoid or mitigate the behavior to the networks.

Trust management systems have been recently introduced as a security mechanism in MANET. In a trust management system, a communicating entity collects evidence regarding competence, honesty or security of other network participants with the purpose of making assessment or decisions regarding their trust relationships [10]. Here ***trust***

means the confidence of an entity on another entity based on the expectation that the other entity will perform a particular action important to the trustor, irrespective of the ability to monitor or control that other entity [9]. For example, a trust-based routing protocol can collect the evidence of nodes misbehaving, form trust values of the nodes and select safest routes based on the trust metrics.

Reputations systems are often seen as a derivation of trust management system. In the reputation system, an entity forms its trust on another entity based not only on the self-observed evidence but also on the second hand information from third parties. One of the influential reputation systems is the CONFIDANT protocol [7].

In the trust management system, reputation system and other trust-based systems, route selection is based on the sending node's prior experience with other nodes in the network. Its opinions about how other entities are honest are constantly changing. Thus, we call the trust management systems and their derivations as ***dynamic feedback mechanisms***. The dynamic feedback mechanisms are usually applied on the current ad hoc routing protocols to rate the trust about other nodes in the network and make routing decisions based on the trust matrix, which is formed according to the evidence collected from previous interactions. By incorporating the dynamic feedback mechanism in the routing protocol, misbehaved nodes are identified and avoided to forward packets. In this way, misbehavior can be mitigated.

1.3 Objective and Sub-tasks

The primary objective of this thesis is to

> *Investigate the state of the art of dynamic feedback mechanisms and protocols; analyze, implement and evaluate CONFIDANT protocols to see how it improves the network performance and what are the side effects of introducing the mechanism to the mobile ad hoc network.*

Following tasks must be done to achieve the primary objective.

1) Study the preliminary knowledge that is required to carry out the main tasks. For example, to understand CONFIDANT protocol one must have some knowledge of Bayesian analysis; to do performance analysis one must learn the methodologies of conducting performance analysis and processing simulation data.
2) Investigate security issues of mobile ad hoc network and current dynamic feedback mechanisms or protocols that are used to solve or mitigate the issues.
3) Investigate and learn how to use the network simulation tool. There are several popular network simulation tools available and we need to choose the one that best suits our needs. The selected network simulator should be studied so that we can use it as platform to implement protocol and conduct simulations.

4) Analyze and implement the CONFIDANT protocol based on Dynamic Source Routing protocol (DSR); evaluate the network performance.

1.4 Structure of the Report

Since we have almost gone through the chapter one, we only briefly present the content of the subsequent chapters in this section.

Chapter 2 Preliminary Information introduces some preliminary information and concepts that will be used in the thesis. Knowledge about DSR, Bayesian estimation, simulation techniques and network simulators are introduced and explained.

Chapter 3 State of the Art presents current research in mobile ad hoc network securities and main solutions. The chapter covers the security issues in mobile ad hoc network, payment system, trust management system, reputation system and cryptographic system.

Chapter 4 Analysis presents the analysis of DSR protocol, CONFIDANT protocol and ns2 network simulator. We also make some assumptions about misbehaved nodes and define name conventions.

Chapter 5 Design first presents the overall framework of CONFIDANT fortified DSR in the view of software architecture. Then more detailed design is explained, including class diagrams and message sequences.

Chapter 6 Implementation and Tests explains the detailed language features and methods used in the implementation. The test method and test cases are also discussed.

Chapter 7 Performance Analysis evaluates the network performance of CONFIDANT fortified DSR. The simulation results are analyzed and compared with that of standard DSR. Several improvements of the CONFIDANT are also discussed and their performances are evaluated. Finally, the characteristics of CONFIDANT are summarized.

Chapter 8 Conclusion and Future Work presents the conclusion and contribution of the thesis. The chapter also describes some work that we do not have time to complete in this thesis but could be investigated in the future.

2 Preliminary information

In chapter one, we have introduced the MANET. This chapter presents other preliminary information and concepts that will be used in other parts of the thesis. Firstly four general modes of routing operations are introduced and compared. The DSR protocol, which is used as underlying routing protocol in the thesis, is explained in detail. Secondly Bayesian estimation and Beta function are explained to pave the way for the analysis of the reputation model of CONFIDANT in the chapter 4. Thirdly some techniques regarding simulation and performance analysis are presented. Finally, several popular network simulation tools are discussed and compared.

2.1 Mobile Ad Hoc Network Routing Protocols

Nowadays there are various routing protocols proposed for the MANET. The most popular ones are DSDV (Destination-Sequenced Distance Vector), TORA (Temporally-Ordered Routing Algorithm), DSR (Dynamic Source Routing) and AODV (Ad-hoc On Demand Distance Vector). These routing protocols can be categorized in different routing operation modes.

2.1.1 Mode of Routing Operations

Proactive vs. Reactive

These two modes concern whether or not nodes in an ad hoc network should keep track of routes to all possible destinations, or instead keep track of only those destinations of immediate interest [8].

Proactive protocols store route information even before it is needed. This kind of protocols has advantage that communications with arbitrary destination experience minimal delay. However it also suffers from the disadvantage that additional control traffic is needed to continually update stale route information. This could significantly increase routing overhead especially for the MANET where the links are often broken.

Reactive protocols, on the contrary, acquire routing information only when it is actually needed. However, the latency of the communication increases tremendously especially when a node communicates to another at the first time.

Source routing vs. Hop-by-hop routing

These two modes concern whether the source node decides the route for a packet to be forwarded to the destination or the intermediate nodes are allowed to decide the next hop until the packet arrives at the destination.

In the source routing protocols, the source node decides the route and puts the route information in the packet header. All the intermediate nodes forward the packet along the route faithfully. This kind of protocols has advantage that the intermediate nodes are not required to maintain the routing information. But it suffers from the disadvantage that the packet size grows because of source routing information carried in each packet.

In the hop-by-hop routing protocols, it is sufficient for the source to know only how to get to the “next hop” and intermediate nodes find their own next-hops until the destination. In contrast to source routing protocols, hop-by-hop routing protocols do not increase packet size but they requires all the intermediate nodes to maintain routing information.

Table 2-1 shows the classification of the routing protocols into the four operation modes we have introduced.

	Reactive	Proactive
Source routing	DSR	
Hop-by-hop routing	TORA, AODV	DSDV

Table 2-1 Categories of routing protocols

Josh Broch et al. has compared the performance of these four routing protocols [11]. The results show that DSR has best throughput performance (above 95%) at all mobility rates and movement speeds. Thus we will use DSR as basic routing protocol in this thesis.

2.1.2 The Dynamic Source Routing Protocol (DSR)

John et al. proposed the dynamic source routing protocol (DSR) [1] which is a routing protocol for use in multi-hop wireless ad hoc networks of mobile nodes. DSR is an on-demand protocol, in which route are only discovered when data need to be transmitted to a node where no route has yet been discovered. The advantage of this on-demand routing protocol is that there are not any periodic routing advertisement and reducing the routing overhead. DSR is also a source routing protocol, allowing multiple routes to any destination and allows each sender to select and control the routes used in routing the packets.

DSR is composed of the two main mechanisms: “Route Discovery” and “Route Maintenance” which are explained below.

Route Discovery

Route Discovery aims at finding routes from a source node to destination. Figure 2-1 illustrates the procedure of Route Discovery. When a source node S wants to send a data packet to some destination node D, it first searches its route cache to find whether there is a route to D. If there is no route to D, then S will initiate a Route Discovery and send out Route Request message which is propagated to all the nodes within its transmission range. At the mean time, it saves the data packet in its send buffer. The Route Request message contains the addresses of source node and destination node, a unique route request identifier and a route record which records all the intermediate nodes that this route request packet has traveled through. S appends itself to the beginning of the route record when it initiates the message.

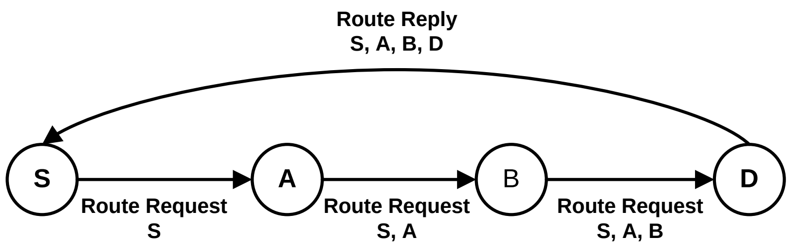


Figure 2-1 Route Discovery

When a node receives the Route Request message, it compares the destination address in the message with its own address to judge whether itself is the destination node. If it is not, it will append its own address in the route record and propagate the message to other nodes.

If the node is the destination node, it will send a Route Reply message to the source node and the message contains the source route record which is accumulated when the Route Request message is forwarded along its way to the destination. When the destination sends the Route Reply, if it uses MAC protocols such as IEEE 802.11 that require a bidirectional link, it just reverse the source route record and use it as route to send Route Reply to the source node. Otherwise it should find the route by searching its route cache or sending out a Route Request which piggybacks the Route Reply for the source node.

When the source node receives the Route Reply message, it puts the returned route into its route cache. From then on all the packets destined to the same destination will use this route until it is broken.

Route Maintenance

Since the ad hoc network is dynamic and the topology of the network changes frequently, the existing routes maintained by nodes in their route cache are often broken. After forwarding a packet, a node must attempt to confirm the reachability of the next-hop node. If the node does not receive any confirmation from the next hop during a certain period of time, it will retransmit the packet. If after a maximum number of retransmission it still does not receive any confirmation, it will think the link to the next hop is broken and will send a Route Error message to the source node.

DSR proposes three acknowledge mechanisms to confirm that data can flow over the link from that node to the next hop:

- Link-layer acknowledgement which is provided by MAC layer protocol such as IEEE 802.11.
- Passive acknowledgement in which a node hears the next-hop node forwarding the packet and thus confirms the reachability of the link.
- Network-layer acknowledgement in which a node sends an explicit acknowledgement request to its next-hop node.

Passive Acknowledgement

Passive Acknowledgement (PACK) is important in CONFIDANT protocol because it is used to detect whether the next hop forwards the packet or drops it. We explain it in detail in this section.

Passive acknowledgement is used with the assumption that:

- Network links operates bi- directionally.
- The network interface is in the “promiscuous mode”.

When a node taps a new packet in “promiscuous mode” after it originates or forwards a packet, it consider it as an acknowledgement of the first packet if both of following check success [1]:

- The Source Address, Destination Address, Protocol, Identification, and Fragment Offset fields in the IP header of the two packets MUST match.
- If either packet contains a DSR Source Route header, both packets MUST contain one, and the value in the Segments Left field in the DSR Source Route header of the new packet MUST be less than that in the first packet.

If no matched packet is found during PACK timeout, the node will consider the link between the next hop and itself is broken and will send Route Error message to the source node.

Additional features

DSR has additional features such as replying to route requests using cached routes, caching overheard routing information, packet salvaging and flow state extension and etc. We will introduce them in section 4.1 and discuss how they will impact the performance of network, how they will interact with CONFIDANT and whether they will be enabled in our simulation.

2.2 Bayesian Estimation and Beta Distribution

Bayesian estimation plays an important role in the CONFIDANT protocol. It is used to model the reputation formation, estimate the behavior of network participants and make decisions. We explain it in this section.

2.2.1 Bayesian Estimation

Bayesian estimation is a statistical procedure which endeavors to estimate parameters of an underlying distribution based on the observed distribution [2]. Given a prior belief of the probability of some event happens, information that is acquired at each observation is update to reflect the added knowledge and to increase the precision of the belief. Equation 2-1 shows the Baye's theorem.

$$P(\theta_i \mid y) = \frac{P(y \mid \theta_i)P(\theta_i)}{\sum_{i=1}^{n} P(y \mid \theta_i)P(\theta_i)} \qquad \textbf{Equation 2-1}$$

Following example explains the meaning of the equation as well as illustrates how Bayesian analysis is used in CONFIDANT to predict the probability whether a node misbehaves or not.

Suppose in the MANET a node *i* has never met node *j* before. *i* has a hypothetic prediction $P(\theta_i)$ about the probability of whether node *j* will misbehave or not. Here θ_i is the ***model parameter*** representing a node misbehaves or behaves well. $P(\theta_i)$ is the ***prior distribution*** which means a probability of θ_i before any data have been observed. After *i* has communicated with *j*, *i* gets observed data *y* about *j*. Then we can know $p(y \mid \theta_i)$, a probability of the data *y* given a know parameter θ_i. However, what we want to estimate is the probability of θ_i given observed information *y*. It is called ***posterior distribution*** and expressed as $p(\theta_i \mid y)$. With Equation 2-1, we can see that $p(\theta_i \mid y)$ can be calculated if $P(\theta_i)$ and $p(y \mid \theta_i)$ are known. After $p(\theta_i \mid y)$ is calculated, it will be used as the prior distribution in the next interaction. This approach of estimating a belief using Bayesian analysis is illustrated in Figure 2-2.

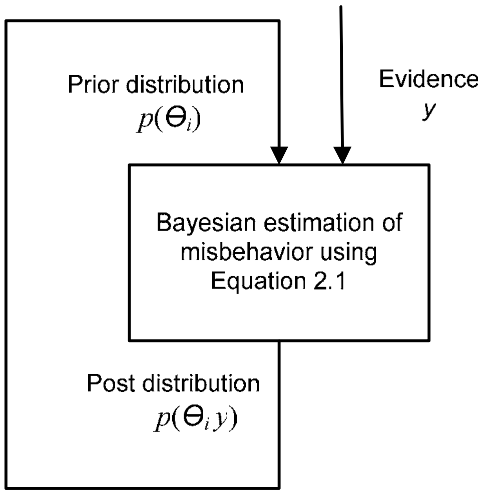


Figure 2-2 Bayesian estimation of misbehavior

2.2.2 Beta Distribution

Given $p(y \mid \theta_i)$, if we want to calculate $p(\theta_i \mid y)$ we still need to know $P(\theta_i)$. $P(\theta_i)$ is the prior which reflects node *i*'s opinion about *j* at the initial situation. CONFIDANT chooses Beta function as the prior distribution. The Beta function can be expressed as follows

$$f(\theta) = \text{Beta}(\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}$$
$$\Gamma(x+1) = x\Gamma(x), \quad \Gamma(1) = 1$$
Equation 2-2

The advantage of using Beta function as prior distribution is that the posterior can also be represented in Beta function. Let *s* represent the number of success and *f* represent the number of failures, the posterior is updated at each observation according to Equation 2-3.

$$\text{Beta}(\alpha,\beta)' = \text{Beta}(\alpha+s,\ \beta+f)$$
Equation 2-3

The characteristic of the Beta function makes implementation easily since only α and β need to be stored to represent the belief. Furthermore, the mean and variance of Beta function can be calculated very easily.

$$\text{E}(\text{Beat}(\alpha,\beta)) = \frac{\alpha}{\alpha+\beta}$$
Equation 2-4

$$\delta^2(\text{Beat}(\alpha,\beta)) = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$$
Equation 2-5

Equation 2-4 is widely used in CONFIDANT to conduct deviation test and decision making which will be discussed in section 4.2

2.3 Performance Analysis Techniques

This section introduces the performance analysis techniques and methodologies that will be used in the performance evaluation.

2.3.1 Factors and Primary Factors

There are many parameters that will influence the simulation results and need to be carefully chosen in the simulations. Some parameters are chosen based on experience values or the conditions of the network we want to simulate. Others need to be tuned to optimize the network performance. We distinguish the two kinds of parameters as follows:

- ***Factors*** are the variables that affect the simulation result and have several alternatives. Normally they are decided based on experience.
- ***Primary factors*** are the factors whose effects need to be quantified. This kind of factors usually needs to be adjusted through simulation.

2.3.2 Data Measurement

The key step of the network performance analysis is to interpret the simulation result and summarize the characteristic of the network. To avoid the inaccurate simulation results due to an extreme scenario, we usually run simulations on several different scenarios. The data set of these simulations are called ***sample***. A single number must be presented to give the key characteristic of the sample and this single number is called an ***average*** of the data.

There are three alternatives to summarize a sample [13]:

- ***Mean*** is obtained by taking the sum of all observations and dividing this sum by the number of observations in the sample.
- ***Median*** is obtained by sorting the observations in an increasing order and taking the observation that is in the middle of series. If the number of the observations is even, the mean of the middle two values is used as a median.
- ***Mode*** is obtained by plotting a histogram and specifying the midpoint of the bucket where the histogram peaks.

Following equations represent how mean, median and mode are calculated.

$$A_{mean} = \frac{\sum_{i=1}^{n} y_i}{n}$$ **Equation 2-6**

$$A_{median} = \begin{cases} Y[n/2+1] & n \text{ is odd} \\ \frac{Y[n/2]+Y[n/2+1]}{2} & n \text{ is even} \end{cases}$$ **Equation 2-7**

where Y is the acending sort of sample y_i: $Y = \uparrow y_i$

$$A_{\mathrm{mod}e} = Y[Max(Frenquency(y_i))]$$
where Y is the list of sample y_i **Equation 2-8**

Different types of sample should consider different ways to calculate average. If the variable is categorical, the mode is the best way to describe the data. If the total of the all data is of interest, mean is a proper index of central tendency. If the histogram is skewed, the median is more representative of the observed data.

2.3.3 Confidence Interval for the Mean

In our performance evaluation, the main objective is to compare the simulation results of CONFIDANT and Standard DSR to see whether there is any performance improvement. However, most simulation results are random in some degree due to the particularity of the node movement scenarios and we cannot tell whether the two systems are different. One way to minimize the random effect is to repeat the simulations with different scenarios as many times as possible and get a large sample space. Unfortunately, due to the time limitation we cannot conduct many simulations. [13] points out that using *confidence interval* we can tell whether the two systems are different with smaller sample space. The confidence interval for the mean can be calculated using Equation 2-9.

$$CI = \left(\bar{x} - \frac{z_{1-\alpha/2} * s}{\sqrt{n}},\ \bar{x} + \frac{z_{1-\alpha/2} * s}{\sqrt{n}} \right)$$ **Equation 2-9**

where CI is confidence interval, $\bar{x}$ is mean, n is sample size
s is standard deviation, $z_{1-\alpha/2}$ is $(1-\alpha/2)-$quantile

If the confidence intervals of the simulation results of the two systems have no overlap, then we can claim the two systems are different and one system is superior or inferior to the other.

2.4 Network Simulators

Nowadays there are many network simulators that can simulate the MANET. In this section we will introduce the most commonly used simulators. We will compare their downsides and upsides and choose one to as platform to implement CONFIDANT protocol and conduct simulations in this thesis.

2.4.1 Network Simulator – ns2

Ns2 is a discrete event simulator targeted at networking research [6]. It provides substantial support for simulation of TCP, routing and multicast protocols over wired and wireless networks. It consists of two simulation tools. The network simulator (ns) contains all commonly used IP protocols. The network animator (nam) is use to visualize the simulations. Ns2 fully simulates a layered network from the physical radio transmission channel to high level applications.

Ns2 is an object oriented simulator written in C++ and OTcl. The simulator supports a class hierarchy in C++ and a similar class hierarchy within the OTcl interpreter [29]. There is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compile hierarchy. The reason to use two different programming languages is that OTcl is suitable for the programs and configurations that demands frequent and fast change while C++ is suitable for the programs that have high demand in speed.

Ns2 is highly extensible. It not only supports most commonly used IP protocols but also allows the users to extend or implement their own protocols. The latest ns2 version supports the four ad hoc routing protocols introduced in section 2.1, including DSR. It also provides powerful trace functionalities, which are very important in our project since various information need to be logged for analysis. The full source code of ns2 can be downloaded and compiled for multiple platforms such as Unix, Windows and Cygwin.

2.4.2 GloMoSim

GloMoSim is a scalable simulation environment for wired and wireless network systems. Currently it only supports protocols for a purely wireless network [12]. It is also built in a layered approach such as OSI seven layer network architecture.

GloMoSim is designed as a set of library modules, each of which simulates a specific wireless communication protocol in the protocol statck. The library has been developed using PARSEC, a C-based parallel simulation language. New protocols and modules can be programmed and added to the library using this language. The latest version of GloMoSim has implemented DSR.

GloMoSim’s source and binary code can be downloaded only by academic institutions for research purposes. Commercial users must use QualNet, the commercial version of GloMoSim.

2.4.3 OPNET Modeler

OPNET Modeler is commercial network simulation environment for network modeling and simulation. It allows the users to design and study communication networks, devices, protocols, and applications with flexibility and scalability [30]. It simulates the network graphically and its graphical editors mirror the structure of actual networks and network components. The users can design the network model visually.

The modeler uses object-oriented modeling approach. The nodes and protocols are modeled as classes with inheritance and specialization. The development language is C.

2.4.4 Comparison

When choosing a network simulator, we normally consider the accuracy of the simulator. Unfortunately there is no conclusion on which of the above three simulator is the most accurate one. David Cavin et al. has conducted experiments to compare the accuracy of the simulators and it finds out that the results are barely comparable [31]. Furthermore, it warns that no standalone simulations can fit all the needs of the wireless developers. It is more realistic to consider a hybrid approach in which only the lowest layers (MAC and physical layers) and the mobility model are simulated and all the upper layers (from transport to application layers) are executed on a dedicated hosts (e.g. cluster of machines).

Although there is no definite conclusion about the accuracy of the three network simulators, we have to choose one of them as our simulation environment. We compare the simulators using some metrics and the results are summarized in Table 2-2.

	Free	Open source	Programming language	DSR implemented
NS-2	Yes	Yes	C++, TCL	Yes
GloMoSim	Limited	Yes	Parsec	Yes
OPNET Modeler	No	No	C	Yes

Table 2-2 Comparison of the three simulators

After comparing the three simulators, we decide to choose ns2 as network simulator in our thesis because

- Ns2 is open source free software. It can be easily downloaded and installed.

- The author of the thesis has used ns2 in another network related course and gotten familiar with the simulation. Ns2 uses TCL and C++ as development languages for which the author has some programming experience.
- The author of the CONFIDANT protocol has conducted simulation on GloMoSim and gotten performance results. We want to do the simulation on a different simulation to form comparison.

3 State of the Art

In this chapter we will introduce the start of the art security solutions in MANET with emphasis on dynamic feedback mechanisms. Firstly, we will present the general security issues/requirements of MANET to pave the way for the future investigation. Then we will discuss the state of the art security mechanisms for MANET such as payment system, trust management system, reputation system, etc. Finally, we will summarize all the security solutions we discussed in this chapter.

3.1 Security Issues in Mobile Ad Hoc Network

Due to lack of central infrastructural and wireless links susceptible to attacks, security in ad hoc network has inherent weakness. In section 1.2 we have discussed the reasons why mobile ad hoc network imposes security challenges that cannot be solved by traditional security mechanisms. In this section, we present the general security properties required by ad hoc network.

Following are general security properties regarding ad hoc network [7] [14] [15] [16].

- ***Confidentiality***: The confdiantiality property is to protect certain information from unauthorized disclosure. The information includes not only the application data that send over the routing protocol, but also the routing information itself as well as network topology and geographical location.

- ***Integrity***: The integrity ensures that the transmitted message and other system asset are modified only by authorized parties. In the routing level, it requires all nodes in the network following correct routing procedure.

  The main challenge of ensuring integrity is that without central infrastructure and powerful computing capabilities, it is difficult to apply existing cryptography and key management systems.

- ***Availability***: The availability property requires that the services or devices are exempt from denial of service, which is normally done by interruption, network or server overload.

  Typical examples or denial of service attack are *radio jamming*, in which a misbehaved node transmit radio to interference other nodes' communications, and *battery exhaustion*, in which a misbehaved node interact with a node for no other purpose than to consume its battery energy.

- ***Authentication***: The authentication property requires that the communication entity’s identification is recognized and proved before communication starts.

- ***Access control***: This property requires restricting resources, services or data to special identities according to their access rights or group membership.

- ***Non-repudiation***: This property ensures that when data are sent from sender to receiver, the sender cannot deny that he has sent the data and the receiver cannot deny that he has received the data.

Mobile nodes may conduct different misbehavior for different purposes. Po-Wah Yau classifies the misbehaved nodes into following categories [15]:

- ***Failed nodes*** are simply those unable to perform an operation; this could be because power failure and environmental events.
- ***Badly failed nodes*** exhibit features of failed nodes but they can also send false routing messages which are a threat to the integrity of the network.
- ***Selfish nodes*** are typified by their unwillingness to cooperate as the protocol requires whenever there is a personal cost involved. Packet dropping is the main attack by selfish nodes.
- ***Malicious nodes*** aim to deliberately disrupt the correct operation of the routing protocol, denying network service if possible.

These four types of misbehaved nodes actually can be categorized in two aspects: whether their misbehaviors are intentional or unintentional, and the severity of the results. Figure 3-1 illustrates the categories.

Severity of the results	Unintentional	Intentional
	Failed nodes	Selfish nodes
	Badly failed nodes	Malicious nodes

Figure 3-1 Categories of misbehaved nodes

3.2 Payment Systems

Payment systems provide economic incentives for the cooperation in MANET. They consider that each node in MANET is its own authority and tries to maximize the benefits it gets from the network. Thus each node tends to be selfish, dropping packets not destined to them but make use of other nodes to forward their own packets. The purpose of payment systems is to encourage the cooperation within the MANET by economic incentives. There are several variations of payment systems proposed.

3.2.1 Nuglets

Nuglets [17] is a virtual currency mechanism for charging (rewarding) server usage (provision). Nodes that use a service must pay for it (in nuglets) to nodes that provide the service. A typical service is packet forwarding which is provided by intermediate nodes to the source and the destination of the packet. Therefore either the source or the destination should pay for it.

There are two models for charging for the packet forwarding service: the ***Packet Purse Model*** (PPM) and the ***Packet Trade Model*** (PTM).

In the Packet Purse Model, the sender pays for the packet. It loads the packet with a number of nuglets when sending the packet. Each intermediate forwarding node acquires some nuglets from the packet that covers its forwarding costs. If a packet does not have enough nuglets to be forwarded, then it is discarded. If there are nuglets left in the packet once it reaches destination, the nuglets are lost.

In the Packet Trade Model, the destination pays for the packet. Each intermediate node “buys” the packet from previous one for some nuglets and “sells” it to the next one for more nuglets until the destination “buys” it.

Either of the two models has advantages and disadvantages. While the Packet Purse Model deters nodes from sending useless data and avoids the network overloading, the Packet Trade Model can lead to an overload of the network and the destination receives packets it does not want. On the other hand, in the Packet Purse Model it is difficult to estimate the number of nuglets that are required to reach a given destination. But the Packet Purse Model does not need to consider this problem.

To take advantages of the two models and avoid the disadvantages, a ***hybrid model*** is suggested. In this model, the sender loads the packet with some nuglets before sending it. The packet is handled according to the Packet Purse Model until it runs out of nuglets. Then it is handled according to the Packet Trade Model until the destination buys it.

3.2.2 Counter

To address the problems encountered by the nuglets approach such as difficulty in estimating pre-load nuglets and possible network overload, another payment approach based on credit counter is suggested [18]. In this approach, the current state of each node is described by two variables b and c, where b is the remaining battery power and c stands for the value of its nuglet counter. More precisely, b is the number of packets that the node can send using its remaining energy and c is the number of packets a node can originate.

A node can originate a number of packets N only when the condition $c \geq N$ holds. When a node forwards a packet, nuglet counter c is increased by one and b is reduced by one. Thus in order to originate packets, each node must earn credits by forwarding packets.

The counter solution requires tamper resistant hardware security module.

3.2.3 Spirit

S. Zhong et al. proposed Sprite [19], a credit-based system for MANET. As opposed to Nuglets or Counter they do not require tamper-proof hardware to prevent the fabrication of payment units. Instead, they introduce a central Credit Clearance Service (CCS). The basic scheme of the system is as follows. When a node receives a message the node keeps a receipt of the message and reports to the CCS when the node has a fast connection to Credit Clearance Service (CCS). The CCS then determines the charge and credits to each node involved in the transmission of a message, depending on the reported receipts of a message.

In this scheme, the sender charges money. A node that has forwarded a message is compensated, but the credit that a node receives depends on whether or not its forwarding action is successful. Forwarding is considered successful if and only if the next node on the path reports a valid receipt to the CCS.

3.2.4 Discussion on the Payment Systems

The payment systems we describe in above sections either assumes a tamper resistant hardware module is available to ensure that the behavior of the node is not modified or requires a central authority server to determine the charge and credit to each node involved in the transmission of a message. Tamper resistant hardware may not be appropriate for most mobile devices because it demands advanced hardware solution and increases the cost of the devices. Lacking of central authority server is right the inherent property of MANET that causes security challenges so it is also not appropriate.

Furthermore, all the approaches described above suffer from locality problems [20] that nodes in different locations of the network will have different chances for earning virtual currency, which may not be fair for all nodes. Usually nodes at the periphery of the network will have less chance to be rewarded.

3.3 Reputation System

Reputation systems have emerged as a way to reduce the risk entailed in interactions among total strangers in electronic marketplace [21]. Centralized reputation systems have been adopted by many on-line electronic auctions such as eBay to collect and store reputation ratings from feedback providers in a centralized reputation database. Decentralized reputation systems used by MANET, on the other hand, do not use centralized reputation database. Instead, in these reputation systems, each node keeps the ratings about other node and updating the ratings by direct observation of the behaviors of neighboring nodes or second hand information from other trusted nodes. [33] identifies three goals for reputation systems:

- To provide information to distinguish between a trust-worthy principal and an untrustworthy principals.
- To encourage principals to act in a trustworthy manner
- To discourage untrustworthy principals from participating in the service the reputation mechanism is present to protect.

Most of the reputation systems in MANET are based on trust management system. Trust is such a subjective and dynamic concept that different entities can hold different opinions on it even while facing the same situation. Trust management system can work without reputation system. For example, a mobile node can form opinion about other nodes by direct experience with the nodes.

We can unify reputation system and trust management system to dynamic feedback mechanisms. Former one is a global reputation system and mobile nodes share their own experiences of interaction with other nodes. The later one is a local reputation system in which mobile nodes rating the trustability of other nodes based on its own observation.

3.3.1 CONFIDANT

CONFIDANT is a reputation system aiming at coping with misbehavior in MANET [7] [22] [23]. The idea is to detect the misbehaved nodes and isolate them from communication by not using them for routing and forwarding and by not allowing the misbehaved nodes to use itself to forward packets. CONFIDANT stands for Cooperation Of Nodes: Fairness In Dynamic Ad-hoc Network. It usually works as an extension to on demand routing protocols.

With CONFIDANT each node has four components: Monitor, Reputation System, Trust Manager and Path Manager. These components interact with each other to provide and process protocol information. Figure 3-2 illustrates the architecture of the CONFIDANT components based on DSR protocol [7].

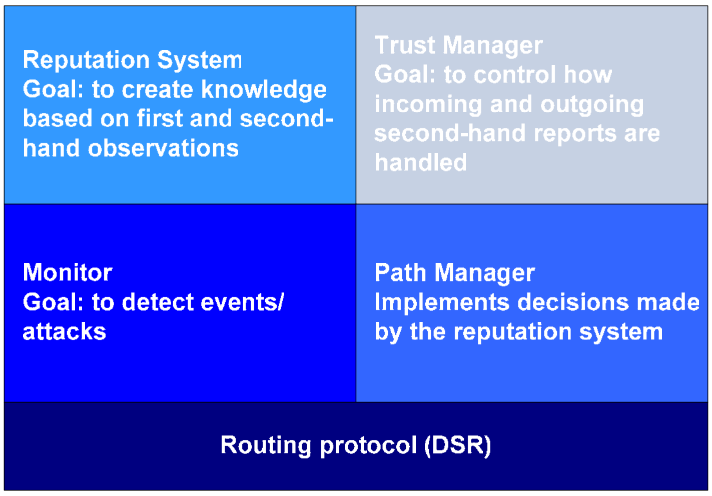

Reputation System
Goal: to create knowledge based on first and second-hand observations
Trust Manager
Goal: to control how incoming and outgoing second-hand reports are handled
Monitor
Goal: to detect events/ attacks
Path Manager
Implements decisions made by the reputation system
Routing protocol (DSR)

Figure 3-2 CONFIDANT components

Monitor is responsible for gathering firsthand information about the behavior of other nodes in the network. This is achieved by observing and detecting various attacks. A typical misbehavior is packet dropping. The monitor detect it by an enhanced Passive Acknowledge mechanism. The monitor can also detect other attacks such as message modification and fabrication through overhearing the packets forwarded by next hop.

Reputation System is the core component of CONFIDANT. It is responsible for maintaining reputation rating about other nodes in the network. The reputation rating about other nodes is updated based on the firsthand information observed by the node or the secondhand information published by other nodes. Reputation System decides whether to accept secondhand information and how much the information is incorporated to update reputation ratings. Based on reputation ratings, Reputation System identifies misbehaved nodes.

Trust Manager maintains the trust rating about other nodes in the network. Trust rating represents a node's opinion about how honest another node is as an actor in the reputation system. It is used as an alternative way to decide whether to accept second hand information. The benefit of using trust is to speed up the detection of misbehaved nodes.

Path Manager performs actions once a misbehaved node is identified, e.g. deletion of paths containing misbehaved nodes, action on receiving request for a rout containing a misbehaved node in the source route, etc.

With CONFIDANT each node collects two major types of information about other nodes which it has communicated or heard about in the network: first-hand information and second-hand information. Based on the information the reputation rating is updated. To have an accurate estimation of misbehavior, Bayesian estimation is employed to form reputation ratings and making other decisions.

CONFIDANT distinguishes trust from reputation. For each node, reputation rating represents how well a node behaves while trust rating represents how honest a node is. Reputation rating is used to decide whether the node is regular or misbehaved, while trust rating is used to decide whether the node is trustworthy or not as a recommender.

Following description illustrates how CONFIDANT works to mitigate the misbehavior in the network.

When a node sends a packet to its neighboring node which is supposed to forward the packet, the node detects whether the neighboring node forwards the packets by listening the packet in a promiscuous mode, which is called Passive Acknowledgement. If it hears the packet is forwarded, it updates the first-hand information and increases the reputation rating about the neighboring node. If it doesn't hear the packet within a certain time, it thinks the neighboring node misbehaves and decreases its reputation rating.

Whenever the reputation rating about another node is updated, the node will identify whether it is misbehaved node or not by comparing the reputation rating with a misbehaved threshold. The identified misbehaved nodes will be reported to Path Manager which will take further actions.

Every node periodically spreads the firsthand information it has collected to its neighboring nodes. If a node receives the published firsthand information and thinks the source of the information is trustable, it will incorporate the information to update the reputation ratings it keeps. It is very important to know that with CONFIDANT a node only forwards or responds to nodes with good behavior. In this way, it isolates misbehaved node by bearing grudges to it.

3.3.2 CORE

Similar to CONFIDANT, CORE (COllaborative REputation mechanism) also provides a mechanism to enforce node cooperation based on a collaborative monitoring technique [24]. However, CORE is different from CONFIDANT in reputation model and the way to spread rumor. Three types of reputations are used in the CORE.

- ***Subjective reputation*** of a target node is the reputation calculated directly from a subject's observation of the target node's behavior.
- ***Indirect reputation*** is evaluated only considering the direct interaction between a subject and its neighbors.
- ***Function reputation*** is the subjective and indirect reputation calculated with respect to different functions such as forwarding a data packet, reply route request.

The final reputation information is combined from the three reputations with different weight associated to the functional reputation value.

CORE consists of two basic components:

- Reputation Table (RT) is a data structure stored in each network entity, keeping the reputation data pertaining to the nodes in the network.
- The Watchdog mechanism (WD) is used to detect misbehaved nodes.

With CORE only positive rating factors are distributed among the entities to avoid a misbehaving entity to distribute false information about other entities in order to initiate a denial of service (DoS) attack.

3.3.3 OCEAN

Both CONFIDANT and CORE use second-hand information which is subject to false accusations and requires a node maintaining trust relationship with other nodes. On the contrary, Sorav Bansal and Mary Baker proposed OCEAN (Observation-based Cooperation Enforcement in Ad Hoc Networks) which only uses direct first-hand observations of neighboring nodes' behavior [25].

With OCEAN, each node has five components:

- ***NeighborWatch*** is used to observe the behavior of the neighboring nodes. It can detect whether the next hop sends the packet successfully or not by Passive Acknowledgement.
- ***RouteRanker*** maintains a rating for each of neighboring nodes. The rating is initialized to neutral and is incremented or decremented on receiving positive or negative events respectively from the NeighborWatch component. Once the rating of a node falls below a threshold the node is put into a fault list and is avoided to be used to forward packets.
- ***Rank-Based Routing*** applies the information from NeighborWatch in the actual selection of routes. To avoid routes containing nodes in the faulty list, an avoid list is added to DSR Route Request Packet (RREQ). On re-broadcasting the RREQ, each node can add its own avoid list to the packet. The nodes receiving the RREQ will check the avoid list in RREQ and decide whether to suppress the packet or reply with Route Reply.
- ***Malicious Traffic Rejection*** rejects all the traffic from nodes it considers misleading so that a node is not able to relay its own traffic under the guise of forwarding it on somebody else's behalf.
- ***Second Chance Mechanism*** allows nodes previously considered misleading to become useful again since a node may "misbehave" due to accidental link error. The misleading node is removed from the faulty list after a fix period of observed inactivity.

3.3.4 LARS

Jiangyi Hu proposed a simple reputation based scheme, called LARS (Locally Aware Reputation System) to mitigate misbehavior and enforce cooperation [26]. Different from global reputation based schemes, with LARS each node *X* only keeps the reputation values of all its one-hop neighbors *N(X)*. The reputation values are updated based on the direct observation of the neighbors. If the reputation value of a neighbor node *M* is under threshold, then *M* is considered by *X* as misbehaved node. *X* will notify its neighbors about *M*'s misbehavior by initiating a warning message. To avoid false accusation, conviction of the uncooperative node is co-signed by *m* different nodes, where *m*-1 is an upper bound on the number of malicious nodes in the one-hop neighborhood. If the warning message is verified, it is then broadcasted to the *k*-hop neighborhood and *M*'s k-hop neighbors become aware of its misbehavior and refuse to serve for it.

3.3.5 Discussion on the Reputation Systems

Although different in reputation model and detailed implementation, all the reputation systems we have introduced have three common parts:

- All the systems detect misbehavior using mechanisms similar to Passive Acknowledgement.
- All the systems maintain reputation ratings about all or part of other nodes in the network and identify misbehaved nodes.
- All the systems react to the misbehaved nodes.

Besides the systems introduced in this section, there are many other reputation systems for MANET. Basically they can be classified into following categories:

- **Global reputation system** in which each node knows reputation value of every other node in the network. This is achieved by exchange indirect reputation message among the network. CONFIDANT and CORE are examples of global reputation system.
- **Local reputation system** in which each node only keeps the reputation value of its neighboring nodes. Instead of distributing reputation value or information periodically, the local reputation systems usually update reputation value based on its own observation.

Jiangyi Hu [26] points out following disadvantages of global reputation systems

- Each node maintains reputation values of every other node, which costs a lot of storage.
- Disseminate reputation information greatly increases the volume of network traffic.
- The decision and incorporating second hand information consumes addition computation.
- The reputation information could be modified, replayed or accidentally lost during transmission.

Hu thinks that the global systems are unnecessary due to above disadvantages. Although the reasons he analyzed may exist, the author of the thesis cannot totally agree with his negative conclusion about the global reputation system. To judge whether a system is good or not depends on how well it can mitigate misbehavior and improve the network performance. If the system can identify the misbehaved node precisely and send the packet successfully at the minimum tries, then the cost of above factors can be compensated.

3.4 Watchdog and Pathrater

Marti et al proposes two techniques that improve throughput in an ad hoc network in the present of misbehaved nodes [27].

The ***watchdog*** method is used for each node to detect misbehaving nodes in the network. When a node sends a packet to next hop, it tries to overhear the packet forwarded by next hop. If it hears that the packet is forwarded by next hop and the packet matches the previous packet that it has sent itself, it considers the next hop behaves well. Otherwise it considers the next hop misbehaves.

The ***pathrater*** uses the knowledge about misbehaving nodes acquired from watchdog to pick the route that is most likely to be reliable. Each node maintains a trust rating for every other node. When watchdog detects a node is misbehaving, the trust rating of the node is updated in negative way. When a node wants to choose a safe route to send packets, pathrater calculates a path metric by averaging the node ratings in the path.

Marti et al implemented the solutions on DSR protocol using ns2 as simulation environment. The simulation result shows the throughput of the network could be increased by up to 27% in a network where packet drop attack happens. However routing overhead is also increased by up to 24%.

3.5 Trust-based Routing

In his master’s thesis, Lennart Conrad developed an improved trust-based routing DSR [14]. With the trust-based routing DSR each node keeps the trust value of all other nodes. Different from most reputation solutions which uses passive acknowledgement to detect whether neighboring node has forwarded a packet or not, trust-based DSR uses an explicit acknowledgement packet sent by the receiver to confirm that the packet has been forwarded by all the nodes along the route successfully. If the sender receives the acknowledgement packet within a timeout, it will increase the trust values of all the nodes along the route. Otherwise it will decrease the trust values. Then a node will choose a most trustful route when it has to send a packet. The main contribution of Lennart’s solution is that he proposed alternative trust value updating and route selection strategies. The simulation results show significant improvement in throughput compared to regular DSR.

3.6 Cryptography

Several cryptography based routing protocols have been proposed based on the modification of existing ah hoc network routing protocols. Ariadne and Key Management System are among them.

3.6.1 Ariadne

Ariadne [28] is a cryptographic based security solution for on-demand routing protocol, for example, DSR. It prevents attacks of compromising routing information or Denial of Service in ad hoc network. Instead of using traditional asymmetric protocols such as RSA, Ariadne primarily uses the TESLA broadcast authentication protocol for authenticating routing messages. The advantage of TESLA is that it is not computing intensive and it only add a single message authentication code (MAC) to a message for broadcast authentication, which is very important for resource limited mobile devices. Ariadne mainly authenticates packets containing Route Request, Route Reply and Route Error to prevent misbehaved nodes changing route information.

Ariadne's authentication mechanisms only check whether the route information is compromised but do not detect whether the messages are dropped. To thwarting such routing misbehavior, it uses feedback about which packets were successfully delivered. The feedback can be received either through an extra end-to-end network layer message, or by exploiting properties of transport layer such as TCP. When there are multiple routes to a single destination, Ariadne sends more fraction of packets along the route with better overall feedback for the nodes in it.

Ariadne assumes

- Network link is bidirectional
- All nodes have loosely synchronized clocks
- One of following keys to be set up: TESLA, pairwise shared secret keys or digital signatures.

3.6.2 Key Management System -- Threshold Cryptography

Using cryptographic schemes to protect routing information and data traffic usually requires a key management service. Zhou et al proposes a key management system which adopts a publish key infrastructure to distribute keys. They use a so-called (n, $t + 1$) $n \geq 3t+1$ threshold cryptography scheme which allows n servers to share the key management responsibility operations, e.g. create a digital signature, so that any $t + 1$

parties can perform this operation jointly but it is infeasible for at most t parties to do so, even by collusion. With threshold cryptography, each server has a public/private key pair. All nodes in the system know the public key of the service and trust any certificates signed using corresponding private key. The scheme divides the private key *k* of the service into *n* shares, assigning one share to each server. Then with *t* + 1 correct partial signatures the combiner is able to compute the signature for the certificate. However compromised servers cannot generate correctly signed certificates because there are at most *t* of them.

3.6.3 Discussion on the Cryptography Systems

Comparing to other security mechanisms such as payment systems and reputation systems, the cryptography systems has the advantage that it can cope well with Denial of Service attack and any form of message modification and fabrication. However, the cryptography by itself cannot effectively deal with packet drop attack. Furthermore, most of the cryptography systems assumes key distribute system available, which is the weakness of mobile ad hoc network.

3.7 Artificial Immune System – an Intrusion Detection System

Slavisa and Jean proposed an artificial immune system [14] [32], which analogs natural Immune System (IS) of human, to detect misbehavior in MANET. The natural IS has two components, innate IS and adaptive IS. The ***innate IS*** is hard-wired to detect and destroy non-self cells that contain or do not contain specific patterns on their surface. Human skin is an example of innate IS. The ***adaptive IS*** detects the non-self cells and “learns” the patterns of the cells and thus can quickly response next time. The idea of the artificial immune system is to map the natural IS elements to a detection system in MANET. For examples, the body of the human is mapped to the entire mobile ad-hoc network; Self cells are mapped to well behaving nodes and non-self cells are mapped to misbehaving nodes; Antigen is mapped to a sequence of observed DSR protocol events recognized in sequence of packet headers, i.e. “data packet sent”, “data packet received”, etc. The simulation shows a good detection capability. However, it is premature to draw general purpose conclusion about the performance of the AIS approach and more investigation need to be done in this area.

3.8 Summary

This section has introduced the general security issues in the MANET and various solutions that have been proposed. The solutions can be classified to several categories, payment system, reputation system, trust-based system and cryptographic system. Each system has its own advantages and disadvantages.

Payment systems serve as an incentive to provide a well-defined service. They are easy to understand and implement. However, to ensure the payment not to be modified by misbehaved nodes, tamper-proof hardware and trusted third parties may have to be required. Furthermore payment systems assume that every node forwards the packets and they are not good at mitigating misbehavior.

Reputation systems aim at encouraging good behavior and punishing misbehaved nodes. They can be further classified into global reputation systems and local reputation systems. Global reputation systems use secondhand information and can speed up detecting the misbehaved nodes. However, distributing the reputation information periodically increases the network overhead. Local reputation systems do not publish the reputation information and mostly use direct information about other participants in the network. They are more lightweight but may be slow at detecting misbehaved nodes.

Trust-based systems are very similar to the local reputation systems in some aspects, except that it only uses firsthand information it has learned in the previous interactions with other misbehaved nodes. Furthermore, trust-based systems are different from local reputation systems in that it never influences the other entities decision by its own reputation ratings.

Cryptographic systems can detect attacks that the other three systems cannot do, e.g denial of service (DoS). However, the systems assumes key distribute system available, which is normally not available in mobile ad hoc network.

We also introduced an intrusion detection system, artificial immune system, which simulates human’s immune system to detect attacks. The mechanism is still in the initial stage and it is not clear whether it is efficient in a real or simulated system.

4 Analysis

In this chapter, we analyze the most important features, models and existing problems in the DSR protocol, CONFIDANT protocol and ns2. These features, models and problems will impact the software design and performance evaluation. We also make assumptions and define the name convention that will be used in the following chapters.

4.1 Analysis of DSR Protocol

Apart from the basic functions of DSR protocol that has been introduced in section 2.1, we are also interested in some additional features of DSR. These features will impact the network performance especially after the integration with CONFIDANT. We will analyze these features and decide whether they will be enabled or not in our project.

In the following section, we call the DSR protocol as standard DSR and the CONFIDANT fortified DSR protocol as CONFIDANT.

4.1.1 The Importance of Route Redundant

All the dynamical feedback mechanisms investigated in chapter 3 rely on inherent redundancies – multiple routes available to a single destination. As long as there are enough good nodes and alternative routes, packets can go around those misbehaved nodes and arrive at a destination. Thus increasing the number of available routes to the same destination is very important.

The route cache is used to store routes in the standard DSR. When a node gets a new route, either by initiating a new route discovery or by overhearing a packet which contains route information, it adds the route into the route cache. When the node wants to send a packet to a destination node, it searches a shortest route to the destination in the route cache. If no route is found, the node will send out a Route Request to find new routes.

With standard DSR, a node selects a shortest route to the destination from the route cache. In CONFIDANT, however, it works in different way. Rather than calculating the shortest hops, a node selects the route which contains no misbehaved nodes. The more alternative routes available in route cache, the more possibility that a node can find a qualified route. Thus when we decide whether to enable a DSR optional feature or not, we use the following criteria:

The additional feature should be enabled if it can increase the number of routes discovered or cached. Otherwise it should be disabled.

4.1.2 Only Forwarding the First Route Request Messag

When a node initiates a Route Discovery, it broadcasts the Route Request message to all its neighboring nodes and these neighboring nodes will append its address to the address list of the message and propagate the copy of the message to their own neighbors. There will be message flooding if the Route Request is forwarded unlimitedly. To mitigate the problem, DSR protocol specifies that a node should only forward the first copy of the same Route Request message it receives.

Figure 4-1 illustrates why the propagation of Route Request should be controlled. The intermediate node A receives a Route Request message for the first time and forwards it to B (see the dashed arrow). B forwards the Route Request to its neighboring nodes (see the dashed-dotted arrow). A receives the Route Request a second time and drops it. If the propagation of Route Request message is not controlled, A will forward the message again to B (see the dotted arrow) and the message will be forwarded endlessly.

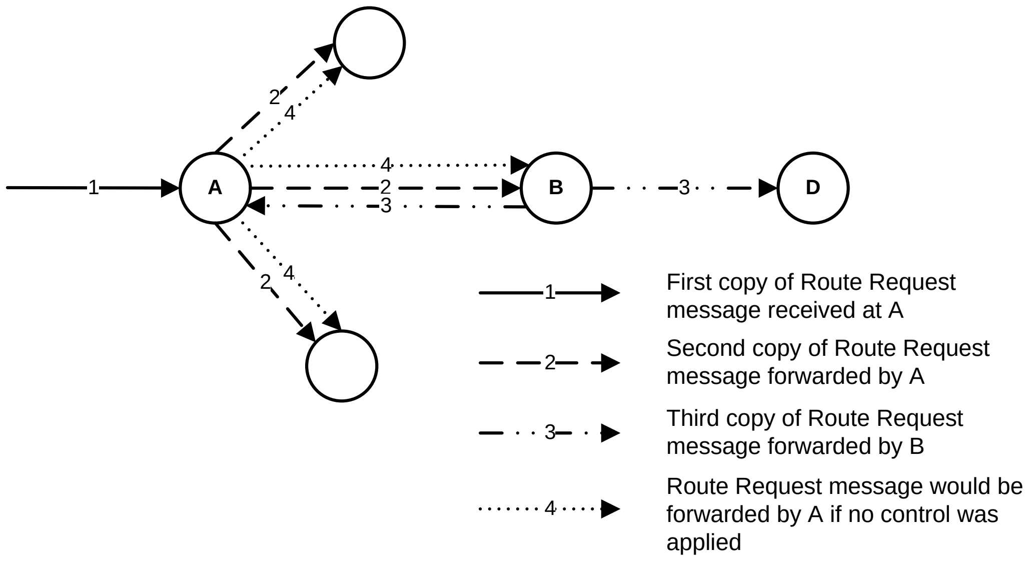

1
A
B
D
2
3
4
First copy of Route Request message received at A
Second copy of Route Request message forwarded by A
Third copy of Route Request message forwarded by B
Route Request message would be forwarded by A if no control was applied

Figure 4-1 Node A only forwards the first Route Request message

Figure 4-2 illustrates another scenario of Route Request propagation control. To discover a route to the destination E, the source node S broadcasts a Route Request message to its neighboring nodes A, B and C. The node D receives the three copies of Route Request message from node A, B and C. (We assume D receives the first copy from B.) If D forwards all the three copies of the Route Request to E, then the number of packets is tripled compared with the case that D only forwards the Route Request it receives from B. Thus in this case, DSR requires that D only forwards the first Route Request and discards the other two.

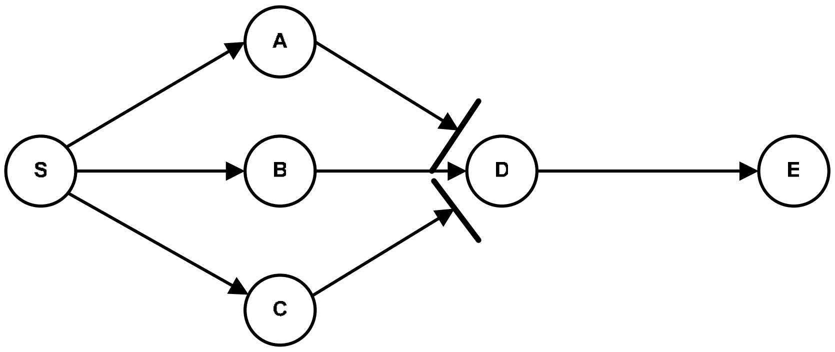


Figure 4-2 Node D only forwards the first Route Request message

This feature, however, limits the number of routes found in each Route Discovery. It will impact the performance of CONFIDANT as explained in the section 4.1.1. If B is a misbehaved node, then the route S → B → C → D will be discarded in route selection and it turns out that S has to initiate another route discovery.

Thus the feature has two contradictory consequences in our project: enabling the feature will increase the number of Route Discovery; disabling the feature will cause message flooding. Since message flooding can cause much more serious problem we think it is reasonable for the feature to be applied.

4.1.3 Caching Overheard Routing Information

In DSR a node overhearing a packet should add all usable routing information from that packet to its own route cache. DSR specifies two ways for a node to overhear the routing information:

1) One way is that when the node is participating in forwarding a packet containing source route option or route error information, the node can update the information into its own route cache. This is a mandatory DSR feature.
2) The other way is an optional feature. If a node is in promiscuous mode and can snoop any packets sent by its neighbors, it will update the route information contained in the snooped packets into the route cache.

The second way can increase the number of routes discovered. This becomes especially important in a network where misbehaved nodes present because the more the routes are available, the more possibilities there are safe routes. Thus we will enable this feature in our implementation.

4.1.4 Replying to Route Requests Using Cached Routes

DSR allows a node receiving a Route Request for which it is not the target to reply with the route found in its own route cache. In the Route Reply, this node appends the source route to the target node obtained from its own route cache after a sequence of hops over which the route request has been forwarded to it. After the node sends out the Route Reply, it will not propagate the Route Request any further.

Figure 4-3 illustrates the scenario of replying to Route Request using cached routes. Source node S initiates a Route Request to find routes to destination D. When B receives the Route Request, it searches its route cache and finds a route B → C → D to the destination D. Then B concatenates the route with the accumulated source route S → A in the Route Request and sends the Route Reply to source node S. Node B will not broadcast the route request further in this case.

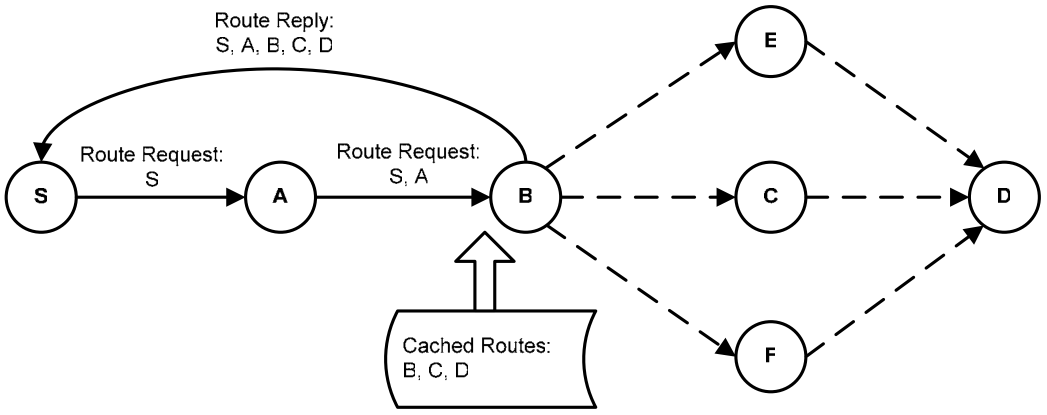


Figure 4-3 Replying to Route Requests using cached routes

This feature, however, will decrease the number of routes discovered. For example, if B does not reply the route request with the cached routes and instead forwards the route request to node C, E, and F. Then we can get two additional routes: S → A → B → E → D, S → A → B → F → D. (As seen in Figure 4-3, the dashed lines indicates the routes could be found). But with this feature we can only get one route. Thus, we will disable this feature.

4.1.5 Route Request Hop Limit

In DSR, each Route Request message contains a “hop limit” that may be used to limit the number of intermediate nodes allowed to forward the Route Request. DSR allows a ring search feature to discover routes. In the ring search mode, the source node sends out a Route Request with hop limit zero, which means only the neighboring nodes can receive the Route Request. If no route was found in the first round of route discovery, the hop limit will be increased to allow the Route Request to be forwarded to greater range of nodes.

This feature can save the routing overhead in case the destination node is one of the neighbors of the source node. However, it will also limit the number of the routes being discovered and thus will be disable in our project.

4.1.6 Packet salvaging

When an intermediate node forwarding a packet detects that the next hop for that packet is broken and if the node has another route to the destination in its route cache, the node will salvage packet by using the new route instead of discard it. To avoid the packet to be salvaged endlessly, a counter is maintained in the packet to count the number of times that it has been salvaged.

The packet salvaging feature can increase the number of packets arriving at the destination. However, since the node which salvages the packet changes the source route header, we should be very careful when using passive acknowledgement to detect the packet modification in CONFIDANT protocol. If a node receives a PACK packet and finds the salvage value is changed, it should not compare the source route with the original one.

4.1.7 Automatic Route Shortening (Gratuitous Route Reply)

DSR allows a source route to be automatically shortened if one or more intermediate nodes become unnecessary. As shown in Figure 4-4, source node S sends a packet to destination node D along the source route S → A → B → C → D. If C snoops the packet forwarded by A and finds itself on the source route but it has not explicitly receives the packet, C can confirm that node B is not necessary to participate in the forward and can be deleted from the source route. Then C sends a gratuitous route reply to notify source node S that there is a shortened route S → A → C → D.

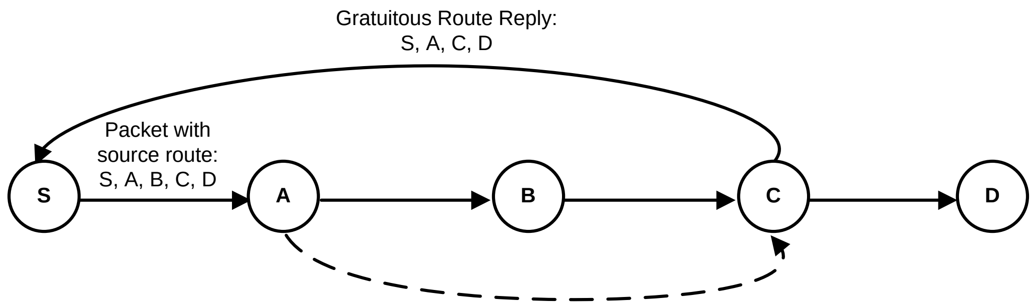


Figure 4-4 Automatic route shortening

The automatic route shortening feature can save the time for a packet to be delivered to a destination. Particularly, in a network where misbehaved nodes present, the shorter route means less possibility that the route contains misbehaved node. Thus we enable this feature in our project.

4.1.8 DSR Flow State Extension

Since the draft version 9, DSR has added a new feature called “flow state” to allow the routing of the most packets without an explicit source route header in the packet. In this feature, when a sending node has discovered a source route, the node can establish a flow along the route which allows each node along the route to forward the packet to the next hop based on the node's own local knowledge of the flow along which this packet is being routed.

The feature significantly reduces the overhead of the protocol because data packet need not carry source route along the way. However, we will not enable this feature in our project. The reason is that the flow state is a new feature and its impact on the CONFIDANT is unknown. CONFIDANT has no specification about how to deal with it. Furthermore we want to compare our simulation result with that of Sonja reported in her Ph.D thesis [7] which does not support flow state.

4.1.9 Other Features

DSR provides other additional features to improve the performance of a MANET. For examples, a node will handle the queued packets in Network Interface Queue and Maintenance Buffer if it detects the next-hop link of the route of the packet is broken; when a node receives a route error it usually spreads the message to other nodes by piggybacking the route error information in the next route request packet. These features have little impact on the performance of CONFIDANT and we will not discuss them in detail.

4.2 Analysis of CONFIDANT Protocol

In this section we will analyze the state chart of CONFIDANT to pave the way for software design. We will discuss the misbehavior detection and data representation that affects the implementation. We will also discuss the network overhead introduced by CONFIDANT which is one of the tasks of performance analysis.

4.2.1 States and Events

As introduced in section 3.3.1, there are five modules in CONFIDANT: DSR, Monitor, Reputation system, Trust manager and Path manager. These modules are interrelated together to provide and process all kinds of information. The modules also interact with DSR to send and receive packets.

Figure 4-5 shows the interactions between the modules of CONFIDANT and DSR in each node. The ovals within each module represent the most important states of the module and the arrow lines indicate events or message between the states.

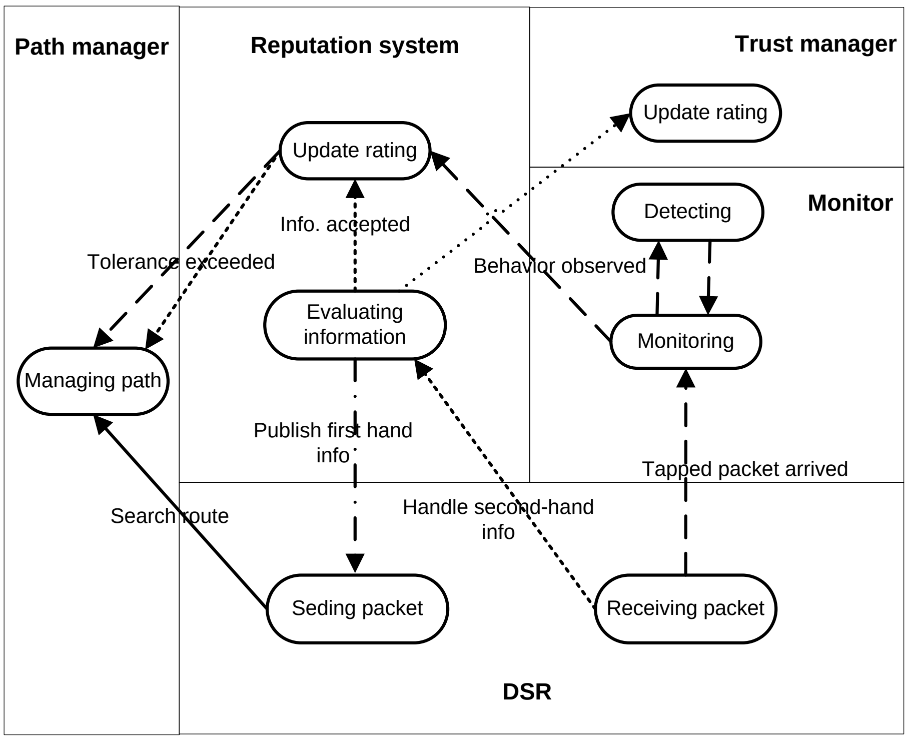


Figure 4-5 Finite state machine in each node

The dashed lines describe how the first hand information is collected. When a node *i* receives a packet in the promiscuous mode from another node *j* within the DSR module, it passes the tapped packet to the Monitor to detect whether it is the PACK packet. If it is, the ratings about *j* will be updated. If the reputation rating is greater than misbehaved threshold, it will inform Path manager to delete all the paths that contains the node *j* from the route cache of node *i*.

The dotted lines describe how second hand information published by the other nodes is handled. As seen in the figure, when node *i* receives published information it passes the information to the Reputation system to decide whether it should be accepted. If the information is accepted, the ratings about node *j* are updated. If the reputation rating after updating exceeds tolerance threshold, all the paths that containing the node *j* will be deleted from Path manager.

The dashed-dotted line describes that a node periodically publishes the reputation ratings it has about other nodes in the network. As seen, reputation system periodically calls the DSR to send out the firsthand information.

4.2.2 Detection of Misbehavior

Misbehavior detection is an important part of CONFIDANT. In this section we will introduce how CONFIDANT detects various misbehaviors. We will also analyze what kinds of misbehaviors cannot be detected by CONFIDANT.

4.2.2.1 Improved Passive Acknowledgement

Section 2.1.2 introduces the Passive Acknowledgement mechanism specified by DSR. CONFIDANT protocol uses the Passive Acknowledgement not only for an indication of the correct reception at the next hop but also to detect whether a node forwards packets that it is supposed to forward or not. In CONFIDANT the passive acknowledgement is improved so that it can have capability to detect more misbehavior types other than dropping packets. When a node overhears a packet, it checks following fields to see whether the packet matches the one it has sent previously.

- IP header: The TTL value must be decremented by one and only one.
- Route reply option: All fields
- Route error option: All fields
- Source route option: If the salvage value is unchanged, all fields except Segs Left; If the salvage flag changed, we only check Type, Last Hop External, First Hop External and Segs Left.
- Forged route error: a node can detect it, if the unreachable address in the route error option is its own.

If any one of above fields does not match, the next hop node is considered misbehaved and its reputation rating is updated in favor of misbehaving. Otherwise, reputation rating is updated in favor of honest.

4.2.2.2 Detectable Misbehavior

Generally the misbehaved nodes can be classified into two categories. One is the selfish nodes which drop packets only for the purposes of saving battery since transmission consumes energy. The other is the evil nodes which may intentionally drop, modify or fabricate packets. The later one could cause much more serious problems. For examples, by modifying a data packet, incorrect information will be sent to destination; by sending forged routing packets, an evil node can create a so-called black hole, a node where all packets are discarded or lost.

Whatever the purposes of the attackers, CONFIDANT can effectively detect drop, modification and fabrication attacks through the PACK mechanism. If a node does not hear the next hop forwards its packet within PACK timeout, it knows the next hop drops the packet. If the packet a node hears is different from the original one, it knows the next hop modifies the packet. If a node hears a packet which indicating the node participated to forward or originate but actually it does not, it know the next hop fabricates the packet. Furthermore CONFIDANT can detect if a node tries to tell big lie by publishing inaccurate information about other nodes. For example, when a node *j* publishes its first hand information about node *k*, it may tell a big lie and publishes opposite information which deviates from *k*'s actual behavior. CONFIDANT can detect this by performing deviation test. When node *i* receives the published information from *j*, it compares the information with the reputation rating it has about *k*. If the difference exceeds the threshold, *i* thinks *j* is lying and reject the information. Furthermore, *i* updates the trust rating of *j* in favor of untrustworthiness. In this way any dishonest will be detected.

In summary, CONFIDANT can effectively detect following types of attack: packet dropping attack, modification attack, fabrication attack and big liars.

4.2.2.3 Undetectable Misbehavior

Although CONFIDANT is effective in detecting and mitigating the security problems such as dropping packets and big liar, there are also some types of attacks that CONFIDANT cannot efficiently cope with. Here we present some of these attacks.

Stealth attack

In the previous section we have discussed how CONFIDANT detects big liars. However, node *j* can make so called stealth attack, in which *j* tells small lies about node *k* so that it can pass the deviation test and gradually change the reputation rating of *k* kept by *i*. CONFIDANT cannot effectively detect the stealth attack.

Collusion

In the network if a certain number of nodes collude and publish similar false information about another node, then the node receiving the information cannot detect this. Figure 4-6 illustrates this. Suppose node *i* has no direct experience with *k* but receives published information from neighborhoods A, B, and C. If the neighborhoods collude and tell false information about *k*, then *i* will gradually believe what it is told and forms false opinion about *k*. When *i* moves to a new place where new neighbors tells correct information about *k*, *i* will not believe them.

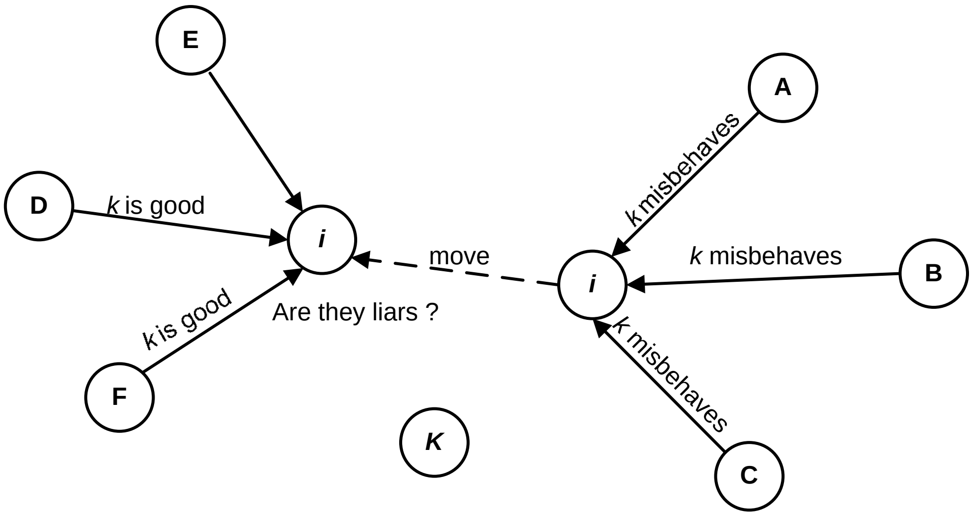


Figure 4-6 Several nodes collude on publishing information about node *k*

<u>Denial of service</u>

If a node sends out a great number of packets to another node in a burst, that node will keep busy with handling the packets so that it cannot send packets of its own. The packets from other nodes also can not arrive at the node because they cannot get the medium to transfer the packets. CONFIDANT cannot detect such kind of attack.

4.2.2.4 False Detection

As stated in section 4.2.2.1, CONFIDANT enhances the Passive Acknowledgement mechanism to confirm the reception of the packet at the next hop and to detect if the next hop fails to forward packets. However, the mechanism of Passive Acknowledge has limitation and can cause false detection. That is, the next hop is regarded as misbehaving because the previous hop fails to hear the next hop forwards the packet but actually it does. The false detection may be caused by transmission collision or the limited transmission range. Following two diagrams depict how the problems happen.

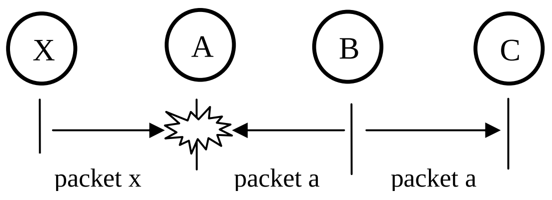


Figure 4-7 Transmission collision

Figure 4-7 illustrates the false detection caused by transmission collision. The node A has sent a packet “a” to B and is expecting that B will forward the packet to C. When B forwards the packet “a” to C, the transmission collides at A with another packet “x” from node X. Then A can not hear the forwarded packet “a” and it thinks B drops the packet.

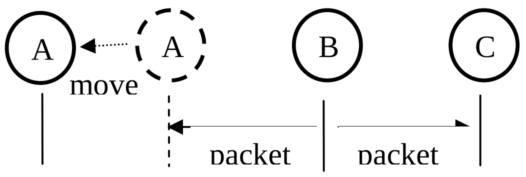


Figure 4-8 Limited transmission range

Figure 4-8 illustrates the false detection caused by limited transmission range. Node A has sent a packet “a” to B and is expecting that B will forward the packet to C. At the mean time A moves out of the transmission range of B. When B forwards the packet “a”, C receives it but A can never hear it. Then A thinks B drops the packet.

4.2.3 Bearing Grudges

One of the major features of CONFIDANT is to mitigate misbehavior by bearing grudges to the nodes identified as misbehaved nodes. On the other hand bearing grudge can also serve as incentive for nodes to behave well and thus improve network performance.

Following are the possible ways to bear grudge to the misbehaved nodes.

(1) Do not select a route containing misbehaved nodes to forward a packet.

(2) Do not forward data packets originated from misbehaved nodes.

(3) Alert the source node when finding a misbehaved node in the source route of the packet that is forwarded.

(4) Do not forward or reply Route Requests originated from misbehaved nodes.

(5) Do not forward or reply Route Request when misbehaved node(s) are present in the source route.

(6) Do not forward or accept Route Error, Gratuitous Route Reply or other routing information originated from misbehaved nodes.

We think item (1), (2), (4), (6) are reasonable ways to bear grudge to misbehaved nodes. However (3) and (5) will not be adopted because the node forwarding a packet should not apply its own subject opinion in deciding whether the packet should be dropped. All the nodes only publish firsthand information but not their opinions! On the other hand, if the node does not forward the packet, it will be considered misbehaved by its previous hop.

4.2.4 Data Representation

According to CONFIDANT protocol, each node stores three types of data about any other nodes it has communicated or heard about in the network. They are

- First hand information $F_{i,j}$: representing the node *i*'s view of node *j*'s behavior as an actor in the network.
- Reputation rating $R_{i,j}$: representing the opinion formed by node *i* about node *j*'s behavior as an actor in the network.
- Trust rating $T_{i,j}$: representing node *i*'s opinion about how honest node *j* is as an actor in the reputation system.

At first glance, it seems that the three types of data are redundant. They are actually different. First hand information only indicates direct observation about other nodes' behavior while reputation rating is a node's opinion of another node which is formed based on first hand information or secondhand information. Furthermore, CONFIDANT distinguishes reputation from trust. The former indicates a node's opinion on how honest another node is as a forwarder and the later indicates how honest another node is as a recommender.

All the three types of data are represented in the form of Beta distribution (α, β) which has been explained in section 2.2.2. They are initialized as (1, 1), which means in the node *i*'s initial view about node *j* there are fifty percent possibility for *j* to misbehave or tell a lie. The updating of the three types of data can be depicted by Figure 4-9:

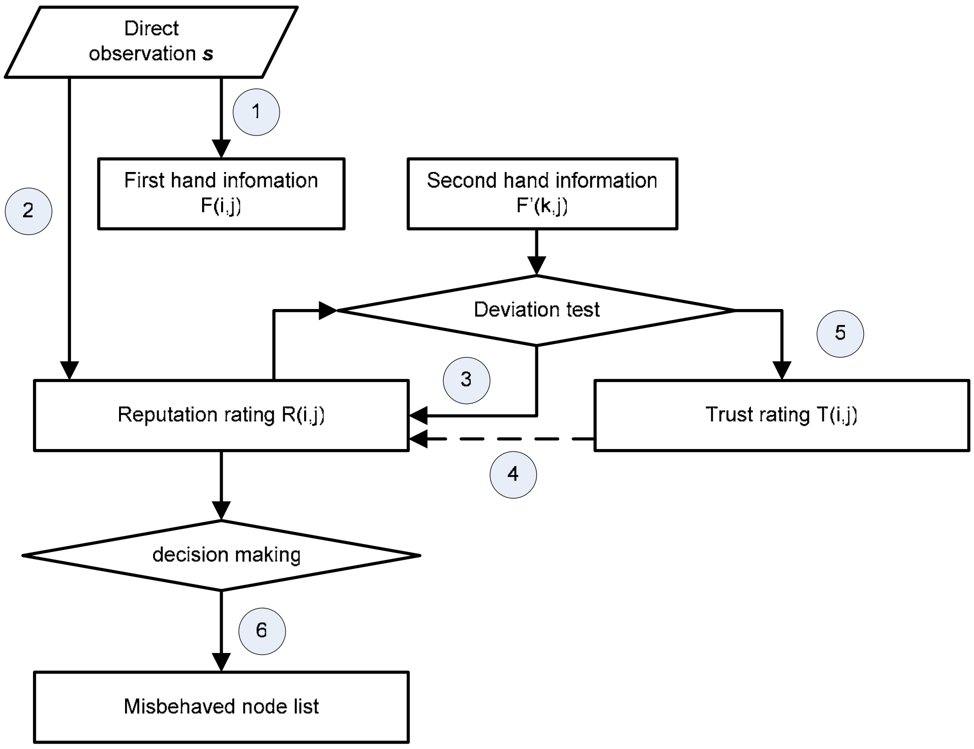


Figure 4-9 The relations of information kept by each node

Updating $F_{i,j}$

The first hand information $F_{i,j}$ is updated based on *i*'s direct observation about *j*'s behavior after each *i*'s interaction with *j* (see the arrow line 1). Let *u* be the weight for the past experience. Let *s* represent the direct observation. *s* = 1 if misbehavior has been observed and *s* = 0 otherwise. $F_{i,j}$ is updated according to Equation 4-1.

$$\begin{cases} \alpha := u\alpha + s \\ \beta := u\beta + (1 - s) \end{cases}$$ **Equation 4-1**

Updating $R_{i,j}$

The reputation rating $R_{i,j}$ is updated upon two events:

(1) When the first hand information is updated (see the arrow line 2 in Figure 4-9).

(2) When the second hand information published by other nodes is accepted (see the arrow line 3 in Figure 4-9).

In the situation (1), the updating of $R_{i,j}$ follows Equation 4-1. In the situation (2), a deviation test is usually done to decide whether to accept the second hand information. The purpose of the deviation test is to reject second hand information that deviates too much from the node's own opinion in case the node that publishes the information tells a lie.

Let the reputation rating of *i* about *j* before updating be $R_{i,j} = (\alpha, \beta)$, the second hand information be $F_{k,j} = (\alpha_F, \beta_F)$. The deviation test is expressed in Equation 4-2.

$$\left|\mathrm{E}(\mathrm{Beta}(\alpha_F, \beta_F)) - \mathrm{E}(\mathrm{Beta}(\alpha, \beta))\right| \geq d \qquad \textbf{Equation 4-2}$$

where *d* is the deviation threshold.

If the result of the deviation test is positive, then the second hand information will be accepted and the reputation rating will be updated according to Equation 4-3.

$$R_{i,j} := R_{i,j} + wF_{k,j} \qquad \textbf{Equation 4-3}$$

where *w* is the weight given to the second hand information.

An alternative way to decide whether to accept the second hand information is to use the trust rating $T_{i,k}$ of node *i* about publisher *k* because trust rating represents a node's opinion about how honest another node is as an actor in the reputation system (see the arrow line 4). Equation 4-4 is used to judge whether a node is trustworthy or not.

$$\begin{cases} \text{trustworthy} & \text{if } \mathrm{E}(\mathrm{Beta}(\gamma, \delta)) < t \\ \text{untrustworthy} & \text{if } \mathrm{E}(\mathrm{Beta}(\gamma, \delta)) >= t \end{cases} \qquad \textbf{Equation 4-4}$$

where *t* is the trustworthy tolerance. (γ, δ) is the Beta distribution form of $T_{i,k}$.

Updating $T_{i,k}$

Trust rating $T_{i,k}$ is updated whenever the deviation test is done (see the arrow line 5 Figure 4-9). The updating follows Equation 4-1 and Equation 4-2 except that the meaning of *s* changes. When updating $T_{i,k}$, $s = 1$ if the result of deviation test is positive, which means node *k* may tell a lie because the information it publishes about *j* is inconsistency with node *i*'s opinion about *j*. $s = 0$ if the result of the deviation test is negative.

Fading and redemption

Fading is introduced to put more emphasis on the recent behavior. Furthermore, fading allows some nodes with bad reputation to redemption. For some nodes their reputations are bad not because they are evil but because they have software or hardware deficiencies so that they work improperly. After the problems are solved, they should be able to improve their reputations by behaving as what route protocol specifies. So the three types of data are faded periodically according to Equation 4-5.

$$\begin{cases} \alpha := u\alpha \\ \beta := u\beta \end{cases}$$ **Equation 4-5**

where *u* is the fading factor and it can be different for three types of data.

Making decision

Each node stores a blacklist of the nodes that are considered misbehaving. A node judges whether another node is misbehaving or not based on its reputation rating about that node. The procedure of making decision is expressed in Equation 4-6. If the mean reputation rating about another node is less than a misbehaved threshold, that node is considered normal. Otherwise it is considered misbehaved.

$$\begin{cases} \text{normal} & \text{if } \mathrm{E}(\mathrm{Beta}(\alpha,\beta)) < r \\ \text{misbehaved} & \text{if } \mathrm{E}(\mathrm{Beta}(\alpha,\beta)) >= r \end{cases}$$ **Equation 4-6**

Where *r* is the misbehaved threshold.

4.2.5 Overhead

CONFIDANT protocol causes overhead in three aspects: message overhead, storage overhead and computation overhead.

Message overhead

With CONFIDANT protocol, a node publishes first hand information periodically with a TTL of 1. The size of the overhead depends on the publishing timer that each node uses as well as the size of the public rating option. The structure of the public rating option is shown in Figure 4-10.

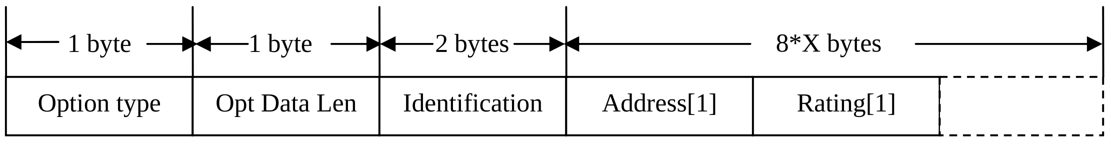


Figure 4-10 Publish rating option

CONFIDANT can cause other implicit routing overhead. Since CONFIDANT sends a packet only when safe routes are found, a node may need to send more Route Requests to discover safe routes.

Storage overhead

Each node stores three types of ratings: $R_{i,j}$, $F_{i,j}$, $T_{k,j}$. about other nodes that it has communicated or heard about. Each of the rating consists of two parameters of Beta function as well as the association to the node ID.

Computation overhead

Since the cost of internal computation in terms of energy consumption is negligible compared to the cost of a transmission or storage, we will not discuss further here.

4.3 Analysis of Network Simulator (ns2)

In this section we will present a problem existing in ns2 and analyze the packet drop reasons of DSR which are the basis of the performance evaluation.

4.3.1 A Performance Problem

The latest ns2 version 2.28 has a serious performance problem when testing with DSR. Searching from the ns-2 email archive [5] shows that the throughput of ns2 after version 2.1b9+ is 10 times worse than that of the previous versions. The throughput is especially bad when there are lots of messages in the network. Evaluation simulations have been conducted for DSR at different packet rates.

The simulations run on ns-2.28 with 50 nodes in the area of 1000 $\times$1000 m^2 at different packet rate. Packet rate means how many packets are sent in one second time. For each packet rate five scenarios were tested and the results are the mean of the throughputs of the five scenarios. The network bandwidth is 2 Mb/s. The simulation results are shown in Figure 4-11.

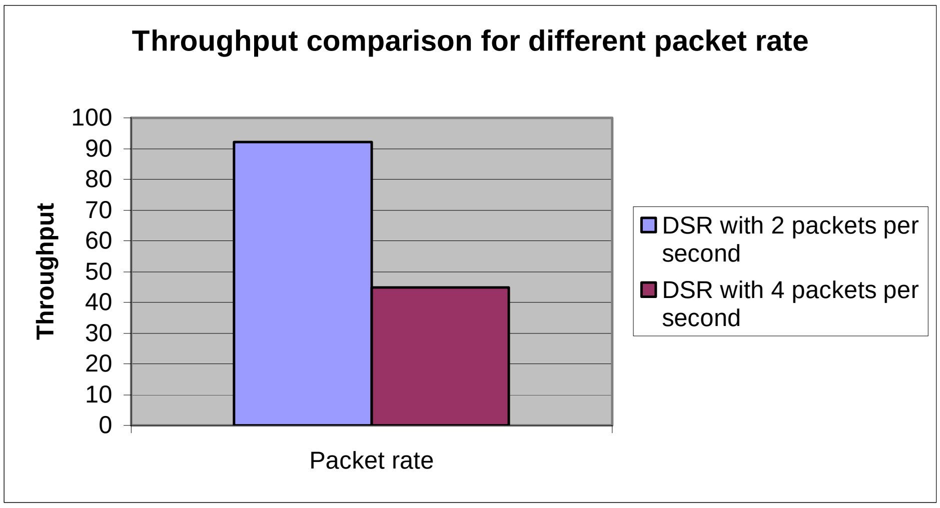


Figure 4-11 Throughput comparison for different packet rate

As seen, the throughput is more than 90% when the packet rate is 2 packets per second. When the packet rate is increased to 4 packets per second, the throughput decreases to less than 45%. By analyzing the trace file, we found that the reason why the throughput deteriorates when packet rate increases is that there are so many packets in the network that most of packets are dropped due to IFQ full. Normally IFQ is full because the network is overloaded so that the link layer cannot grab the transmission medium to send out the packets buffered in the queue. But in our tests the maximum data packets is 6.4k (2 p/s × 64 bytes × 50 nodes) and it cannot exceed 2Mbs bandwidth even plus routing overhead is counted. Currently no one could explain this and the investigation of the reason is out of the scope of the project.

This performance problem could more seriously impact the throughput of CONFIDANT because the protocol introduces overhead by publishing first hand information and send additional routing packets as analyzed in section 4.2.5. Thus our compromised solution is to use the lower packet rate to mitigate the impact.

4.3.2 Packet Drop Reasons

The main task of the performance analysis is not only to interpret the simulation results but also to explain why throughputs are improved or decreases. In most cases we need to investigate where and why packets are dropped. By investigating the ns2 documents, and source code, we summarize the several types of packet drop that play important roles in the simulation. These packet drop reasons are listed in Table 4-1.

Category	Reason	Description
Routing layer drop	Send buffer timeout	Every data packets are saved in the send buffer before they're sent out. If a packet has not been sent out after certain timeout, it will be dropped. In most cases, a packet is timeout because no routes are found within the timeout.
	No route	If a packet is undeliverable (e.g. due to link error) and has been salvaged too many times, DSR will consider there is no route available and drop it.
	TTL reaches zero	It the TTL value in a packet reaches zero, the packet will be dropped.
	Drop by misbehaved nodes	A misbehaved node could drop data packet, route reply, route error.
Link layer drop	IFQ full	Packets are buffered in a network interface queue before they are sent out. If too many packets are generated before previous ones are sent, packets will be dropped.
	ARP full	Before a packet is sent, the MAC address of the destination node must be searched by ARP. If the ARP is full, the packets depending on those nodes will be dropped.

Others	Simulation terminate	Packets that are saved in send buffer of routing layer and IFQ will be dropped when the simulation ends

Table 4-1 Packet drop reasons in ns2

4.3.3 Name Convention

In CONFIDANT protocol, a node classifies the other nodes in the network either as misbehaving or honest based on its reputation rating about those nodes. The identified misbehaved node may actually be a normal one. To avoid confusion when we describe the nature of the nodes, we define following name conventions about the nodes in the aspects of reality and opinion.

We define nodes in reality as follows:

- Evil node: a node that intentionally drops packet.
- Normal node: a node behaves as the routing protocol specifies.

We define nodes in the opinion of other nodes as follows:

- Misbehaved node: a node is considered misbehaving by another node. A misbehaved node may be actually a normal node but is misidentified.
- Good node: a node is considered normal by another node. A good node may be actually an evil node but is misidentified.

4.4 Assumptions

4.4.1 Assumptions about Mobile Ad Hoc Network

Since the CONFIDANT protocol relies on Passive Acknowledgement mechanism to detect nodes that fail to forward packets, we assume following items about the mobile ad hoc network.

- The wireless communication between any two nodes is bi-directional.
- The network interface of any node is in the promiscuous mode.

In this way, a node is able to find out whether the next node forwards the packet if both of them are still in the transmission range of one each.

4.4.2 Assumptions about Misbehaved Nodes

As discussed in section 4.2.2.2, CONFIDANT can detect dropping attacks, modification attacks and fabrication attacks. Due to the time limitation, we only simulate dropping attacks in this project.

In the dropping attack, an evil node could drop all the packets it is supposed forward or destined to itself. It could also be partial dropping, which is restricted to specific types, e.g data packets or routing packets containing Route Error. When we simulate the evil nodes, we want to maximize the bad effect of evil nodes in the network and see whether CONFIDANT can mitigate the problems. Based on this guideline, we make following assumptions about evil nodes.

- The primary interest for evil nodes is to drop data packet since dropping data packet can have more serious consequence than dropping routing packets in general. The purpose of the routing protocol is to enable the data packets exchange between any communication ends in the network. Dropping data packets will also decrease the network throughput significantly.
- Evil nodes drop Route Replies or Route Errors that are not destined to them.
- Evil nodes forward Route Requests. Dropping Route Requests is meaningless for evil nodes if its main target is to drop the data packets. In the mobile ad hoc network where dynamic routing protocol is used, the source node can often find alternative routes to the destination. Thus blocking a route by dropping Route Request may lead to opposite consequence: the data packets will be forwarded by safe routes. Figure 4-12 explains the scenario.

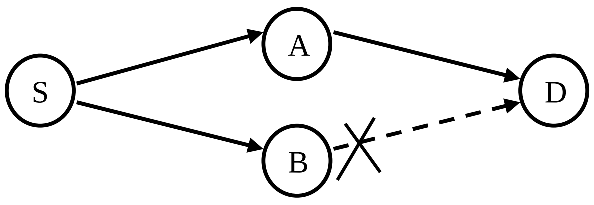


Figure 4-12 A misbehaved node drops Route Request

  Node S wants to send data packet to destination node D. It initiates a Route Request which is broadcast to its neighbors A and B. Suppose B is evil node. If B drops the Route Request, the route S → B → D would never be returned to S. Instead route S → B → D will be used. Thus B will not have chances to receive data packets from S. Since the primary interest for evil nodes is the data packet, we assume B should not drop Route Request.

- Evil nodes reply Route Request destined to themselves because their aim is to drop messages sent to other nodes but they don't want to lose any message destined to themselves.

4.5 Summary

This chapter serves as the software requirement analysis in the view of software development process. In the chapter we have analyzed DSR protocol, CONFIDANT protocol and ns2 simulator regarding the problems that will impact our software design, implementation and performance evaluation.

The analysis of DSR focuses on deciding whether to enable the additional features or not. First of all a criteria is set saying that the enabled features should increase the cached routes as much as possible. Then each additional feature is discussed based on the criteria. Table 4-2 summarizes our analysis about the features.

Features	Enabled/disabled
Only forwarding the first Route Request message	Enabled
Caching overheard routing information	Enabled
Replying to Route Requests using cached routes	Disabled
Route request hop limits	Disabled
Packet salvaging	Enabled
Automatic route shortening	Enabled
DSR flow state extension	Disabled

Table 4-2 Summary of DSR additional features

We then analyzed the CONFIDANT protocol. A state machine is given to describe the states and events in CONFIDANT components. The attack detection mechanism is introduced and a discussion about what kinds of attack can be detected or cannot be detected is elaborated. CONFIDANT maintains three types of information so we analyzed the mathematical representation of the information.

Ns2 is the development and simulation environment of our project. We discussed the potential problem existing in ns2 and proposed a solution. The packet drop reasons are also investigated for future use.

Finally, we make some assumptions and name conventions which will be used in the thesis.

5 Design

This chapter presents the software design of the CONFIDANT forties DSR. First an overall architecture is introduced. Then the class diagrams of CONFIDANT modules, DSR modules and the combination of them are explained in detail. The class diagrams present the static view of the software. We also use sequence chart to describe the dynamic behavior of the modules. Finally we present the design of trace and parser which serves as utility functions in the project.

5.1 Framework and Modules

CONFIDANT fortified DSR protocol is composed of several modules that interact with one another to mitigate misbehavior and improve the network performance. We have analyzed the state machine of the modules in section 4.2.1. Here we present an architecture overview of the modules which gives better description of the component layers of the software and the relationship between them.

Figure 5-1 shows the major modules of CONFIDANT fortified DSR.

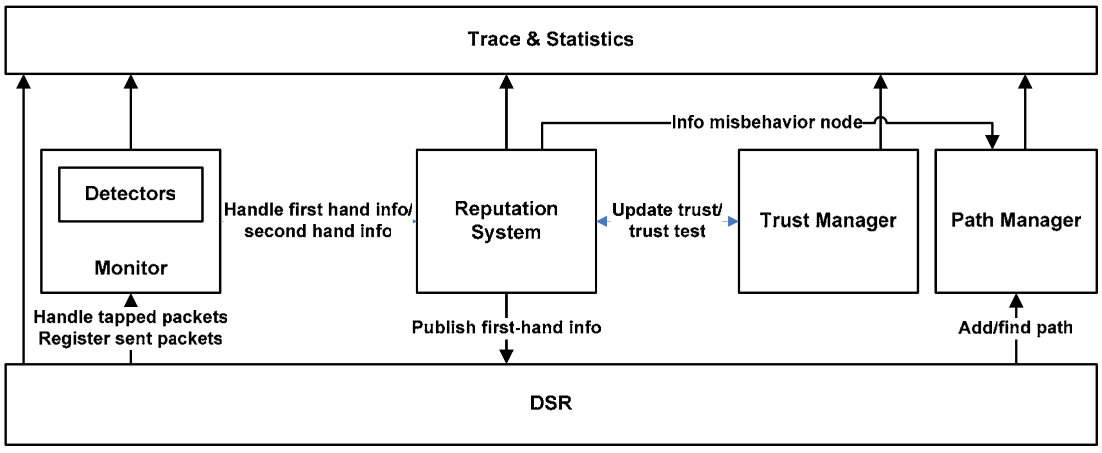


Figure 5-1 Overview architecture of CONFIDANT

Monitor is responsible for registering packets which are sent by a node and supposed to be forwarded by the next hop, detecting whether the PACK packet contains any form of attack and whether next hop drops packet. It also reports the first hand information to Reputation System.

Reputation System maintains the reputation ratings of other nodes based on the self-observed first-hand information as well as second-hand information published by other nodes. It is responsible for identifying the misbehaved nodes and reports them to Path Manager.

Trust Manager maintains the trust ratings of other nodes and decide whether these nodes are trustful when they act as recommenders in the network. It provides an alternative way to decide whether to accept the incoming second-hand information.

Path manager is responsible for maintaining cached routes and selecting a safe route which contains no misbehaved nodes. It also takes actions when misbehaved nodes are identified, e.g. remove the routes containing misbehaved nodes from cache.

DSR is the underlying dynamic source routing protocol. It is responsible for discovering the routes, sending and receiving packets in the routing level and tapping packets in the promiscuous mode.

Trace & statistics are utility functions for logging and statistics. The logged information is the simulation result that will be analyzed in performance evaluation.

5.2 Data Structures

Figure 5-2 is the class diagram of CONFIDANT which is designed based on the architecture displayed in Figure 5-1. As seen, each module of CONFIDANT in Figure 5-1 corresponds to one or more classes in Figure 5-2. In the following sections we will explain in details how the classes are designed for the functionalities of each module.

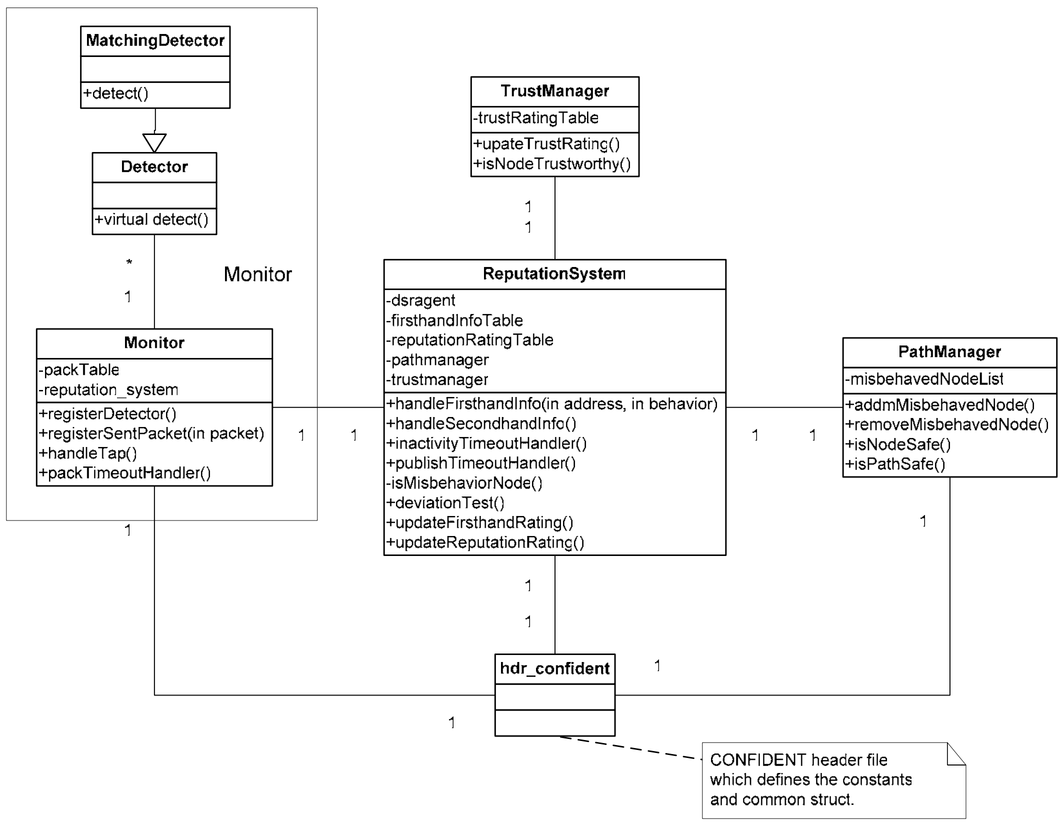


Figure 5-2 Class diagram of CONFIDANT

5.2.1 Monitor

We divide Monitor into two sub-modules. One is the core monitor which serves as the interface to DSR and controls the main logic within the module. The other one is the detector which is responsible for detecting various attacks. As shown in Figure 5-3, Detector class is an abstract class which provides the interface to detect attacks. It has several inherited classes each of which detects different type of attacks.

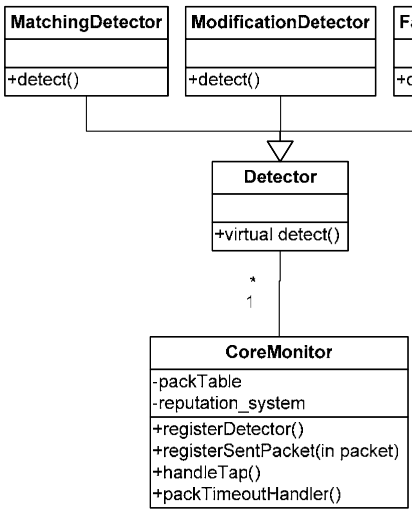


Figure 5-3 Class diagram of monitor module

The classes work in following way. When the program starts, the core-monitor is instantiated. Then instances of different detectors are registered at the core-monitor. When a node sends a packet, it registers the packet in the core-monitor so that the monitor knows that the node wants to verify whether next hop forwards the packet. When the core-monitor receives a tapped packet, it checks whether it is a PACK packet. If it is, the core-monitor calls the detectors to detect whether there are any forms of attack.

The advantage of creating polymorphic detector classes instead of using a single comprehensive Detector class is to make the system extendable. Different types of attack can be added easily without causing a lot of changes in the existing system.

Apart from handling tapped packets heard from next hops, Monitor also keeps a PACK timeout for each registered sent packet and periodically checks whether next hops fail to forward the packet. If no PACK data is heard within PACK timeout then the Monitor will consider the next hop drops the packet and report this information to the Reputation System.

In ns2, every packet is associated with a so-called *uid*, which is the abbreviation of unique identification. We utilize the *uid* to judge whether a tapped packet is the PACK packet that the node is waiting for. When a node sends a packet to its next hop, it registers the packet in Monitor. When the node receives a packet from next hop, it compares the *uid* of the received packet with that of the registered one. If the *uid*s are the same, it thinks the received packet is the PACK packet. Otherwise, it is not.

5.2.2 ReputationSystem

The class ReputationSystem corresponds to Reputation System module. The primary function of the class is to handle the firsthand information given by Monitor and second hand information published by other nodes. The class is also responsible for implementing mathematical models for firsthand information ratings, reputation ratings, Bayesian mean and deviation test.

ReputationSystem handles two timeout events. One is the inactivity timeout event for which firsthand information and reputation ratings are faded according to fading factor. The other is the publishing timeout event for which the firsthand information is published.

Last but not the least, after each updating of a node's reputation rating, ReputationSystem examines the mean reputation value of the node and determine whether it is misbehaving. If any misbehavior node is identified or redampted, the information will be reported to PathManager.

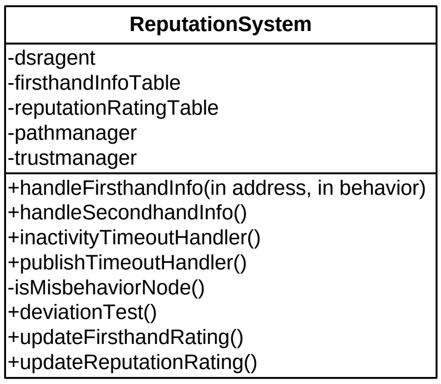


Figure 5-4 ReputationSystem class

5.2.3 TrustManager

The class TrustManager corresponds to TrustManager module. The primary function of TrustManager is to provide an alternative way to decide whether to accept second hand information. The class maintains trust ratings about other nodes. The rating indicates whether a node is trustworthy to be a recommender.

The class works in following way. Whenever ReputationSystem conducts deviation test about node *i*, TrustManager updates the trust rating of *i*. If the result is positive, the rating is updated in favor of trustworthiness. Otherwise, it is updated in favor of untrustworthiness. When node *i* receives secondhand information about node *j*, ReputationSystem queries about whether *j* is trustworthy from TruatManager and then decides whether to accept it.

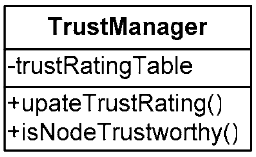


Figure 5-5 TrustManager class

5.2.4 PathManager

The class PathManager corresponds to Path Manager module. It maintains the route cache of DSR. It provides interface to ReputationSystem so that reputation system can inform it when misbehaved nodes are identified or redempted. Since the Path Manager deals with cached route very closely, we extend PathManager from RouteCache class with additional methods (see Figure 5-6).

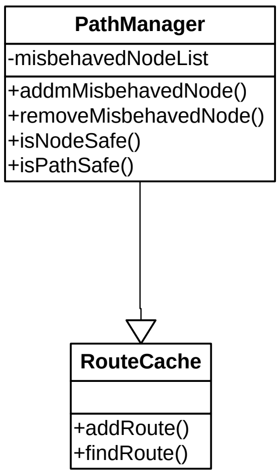


Figure 5-6 PathManager class

CONFIDANT requires that when a misbehaved node is identified, Path Manager should delete all the routes containing the misbehaved nodes. Instead of deleting routes immediately, in our design we mark the routes and keep them in the cache until route cache is full. When a node searches a route in the route cache, Path Manager only selects safe routes which contains no misbehaved nodes.

5.2.5 Existing DSR Components in ns-2

The simulator version ns2.28 that we are going to use already contains the implementation of DSR. The existing DSR components are expressed in Figure 5-7.

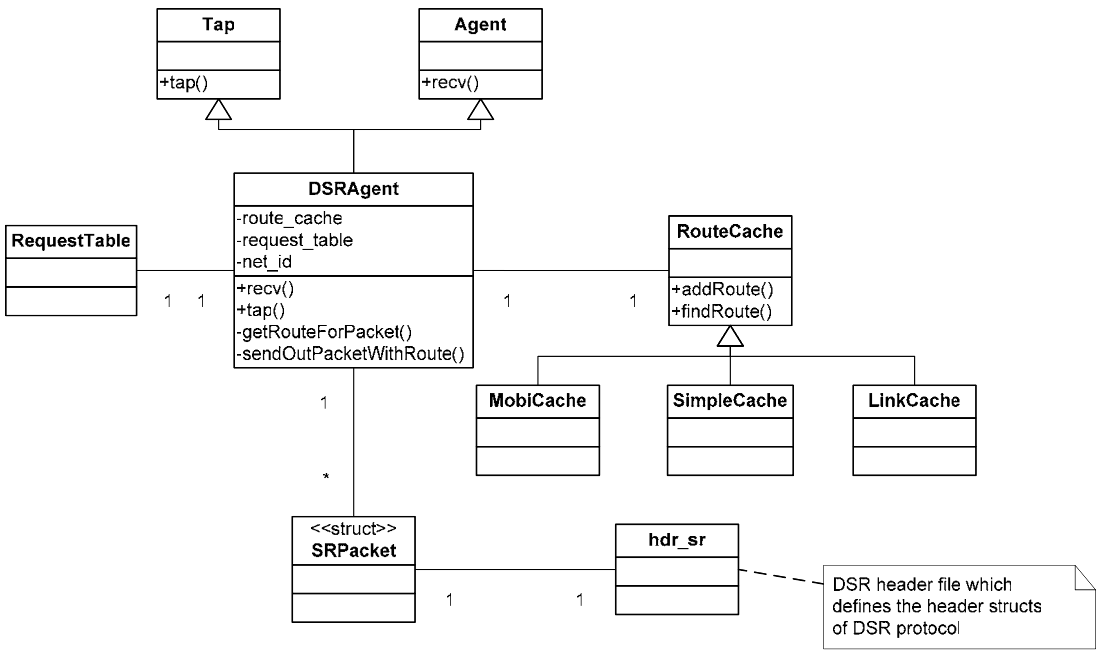


Figure 5-7 Class diagram of DSR in ns-2

As shown in the Figure 5-7, the core class of the DSR module is DSRAgent class. Each node has one instance of DSRAgent. The class inherits from two parent classes.

- The Agent class, through which the DSR is hocked to the ns-2 and runs as routing protocol.
- The Tap class, through which DSR can receive the packets in the promiscuous mode.

Each DSRAgent instance contains one instance of RouteCache. The RouteCache is an abstract class from which various route cache classes inherit to apply different route maintenance strategies. Mobicache is the default route cache of DSR.

Each DSRAgent instance also contains one instance of RequestTable. As we analyzed in section 3.1, a node should only forwards the first copy of same Route Request message it receives. The purpose of RequestTable is to judge whether the node has received the same Route Request before. It records the Route Request packets that a node has processed so that the same Route Request will not be forwarded many times.

Hdr_sr is a header file containing the definitions of all types of DSR option headers.

SRPacket wraps the structures of routing packets or data packets and provide easy access method to get and set the source route.

5.2.6 Combining CONFIDANT and DSR

The combined class diagram of CONFIDANT and DSR can be seen in the Figure 5-8. CONFIDANT interfaces with DSR in the following situations.

- When DSR sends out a packet, it registers the packet in Monitor by calling Monitor->registerSentPacket(packet).
- When DSR taps a packet, it calls Monitor->handleTap(packet) which compares the tapped packet with the ones that registered. The packets are uniquely identified by the *uid*. If the *uid*s of the two packets are the same the tapped packet is considered a PACK packet and the original sent packet is removed from the buffer. Monitor then further call Detectors to detect whether there are any forms of misbehavior.
- When DSR receives a packet which contains first hand information published by other nodes, it passes the information to ReputationSystem by calling handleSecondhandInfo(info).
- When ReputationSystem wants to publish first hand information, it passes the information to DSR. DSR then creates a packet containing the publishing header option and sends out the packet.
- When DSR needs a route to send packet to destination, it calls PathManager->findRoute() which searches the route cache to find a route containing no misbehaved node. If no safe route is available, DSR will initiate a Route Discovery.

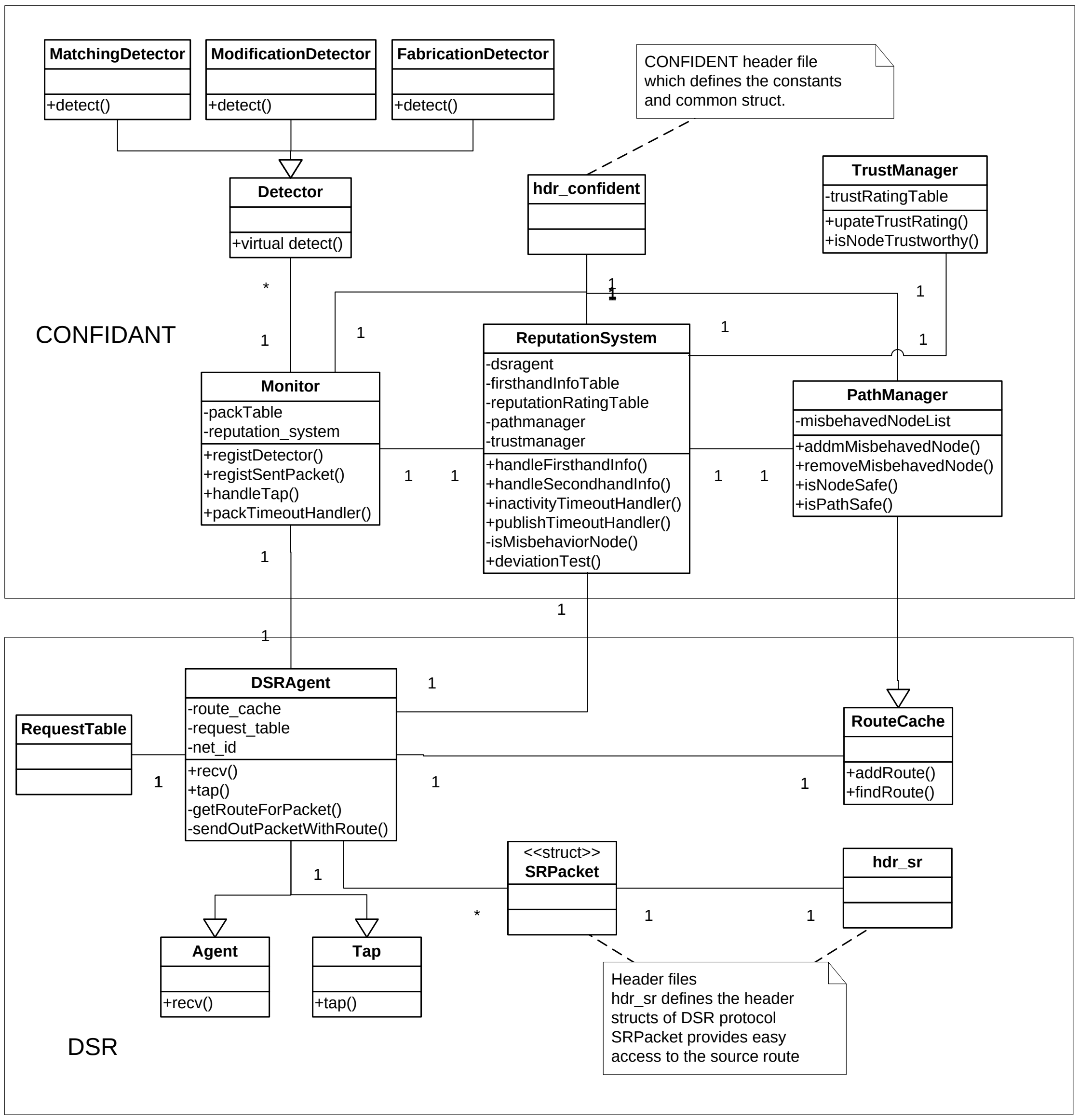


Figure 5-8 Combined class diagram of CONFIDANT and DSR

5.3 Dynamic Behavior

In the CONFIDANT protocol, we concern about following major behaviors:

- Handle first hand information
- Publish first hand information
- Handle second hand information published by other nodes
- Bear grudge to misbehaved nodes

We will explain these scenarios using sequence chart in this section.

5.3.1 Handle First Hand Information

Figure 5-9 illustrates how the first hand information is collected. Three parts of message sequences are involved in this scenario. They are:

- Register the packets sent
- Handle tapped packets
- Handle PACK timeout

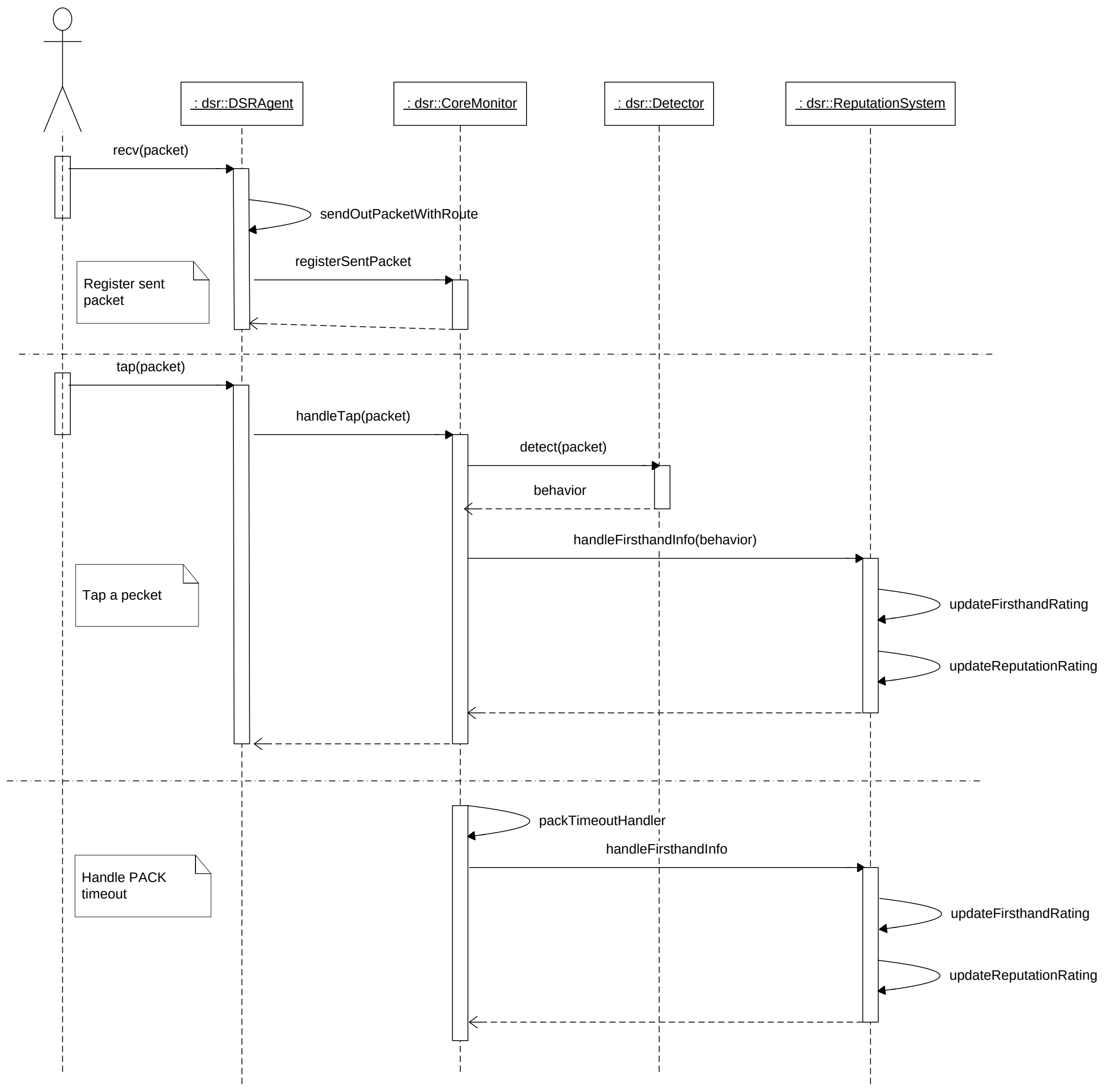


Figure 5-9 Sequence diagram of handling first hand information

.

Register the sent packets. When a node sends a packet, if it wants to check the passive acknowledgement from the next hop, it registers the packet in the monitor by calling Monitor->registerSentPacket(packet) so that the packet can be compared with the PACK packet later.

Handle tapped packets. When a node taps a packet in the promiscuous mode, it calls Monitor->handleTap(packet) to check whether it is a PACK packet that the node is expecting. The Monitor searches the packTable and if it finds a packet with the same uid as that of the tapped packet, it considers it as a PACK packet. It then calls Detector->detector(packet) to check whether the packet is modified maliciously and gets the behavior of the next hop. Finally it calls the methods of ReputationSystem to update the first hand information rating and reputation rating.

Handle PACK timeout. If the Monitor does not detect the PACK packet within certain timeout, it will consider the next hop as misbehaved and call ReputationSystem to update the first hand information and reputation rating.

5.3.2 Publish First Hand Information

Figure 5-10 illustrates the scenario of publishing first hand information.

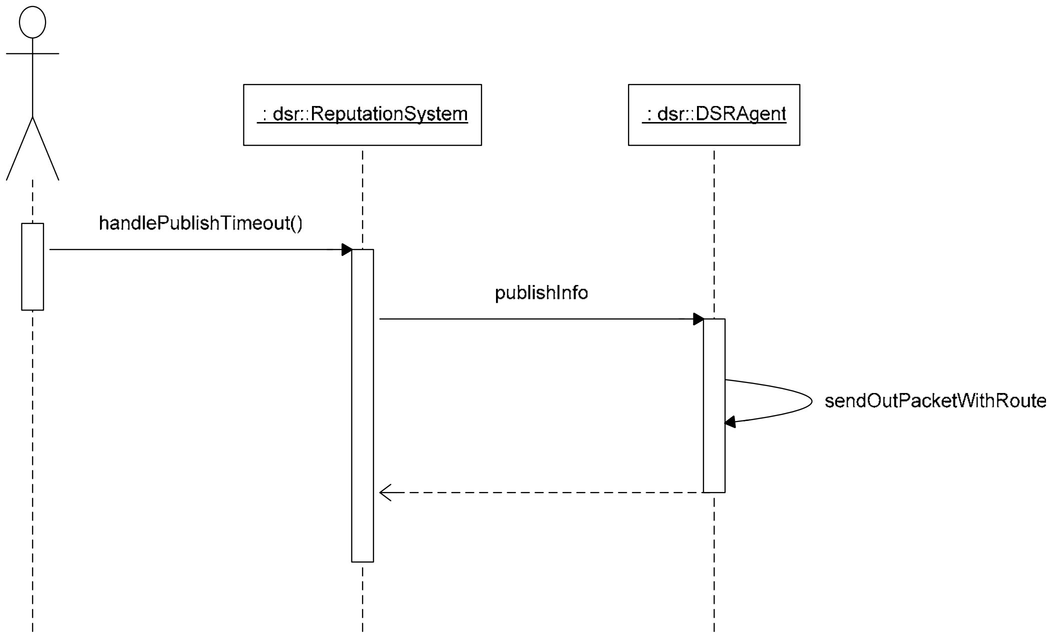


Figure 5-10 Sequence diagram of publishing first hand information

In this scenario, each node periodically broadcasts its firsthand information about other nodes to all the neighboring nodes. When PUBLISH_TIMEOUT expired, the ReputationSystem call the DSRAgent to send out the publishing firsthand information packet. The TTL of the packet is set 1 so that only the neighboring nodes can receive the information and avoid flooding the packets to the whole network.

5.3.3 Handle Second Hand Information

Figure 5-11 illustrates the scenario of handling second hand information.

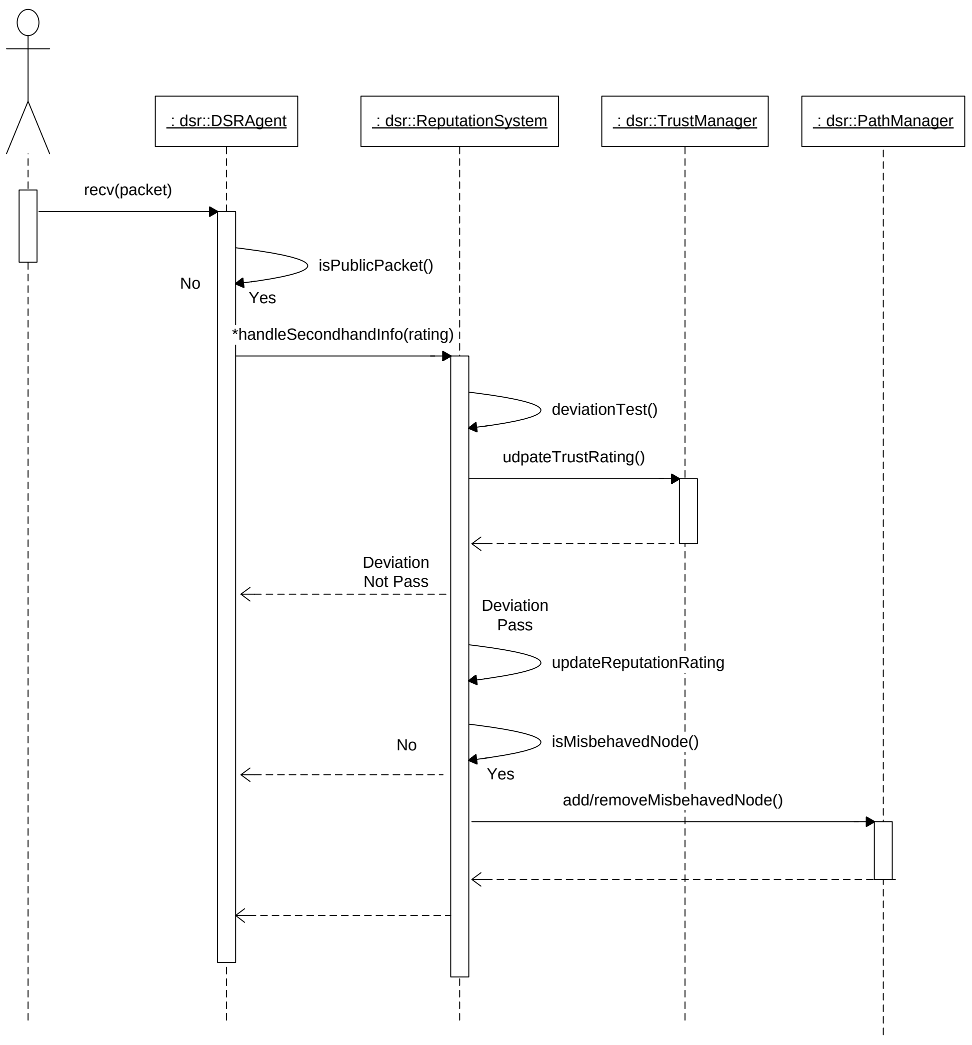


Figure 5-11 Sequence diagram of handling second hand information

When DSRAgent receives a public packet, it gets the firsthand information about several other nodes. It calls ReputationSystem.handleSecondhandInfo(rating) several times to handle each of the information. ReputationSystem first conducts deviationTest(). If the result of the deviation test is positive, the information will not be acceped. Otherwise, the reputation rating of the node will be updated. Then the reputation mean value will be compared with the misbehaving threshold to judge whether it is a misbehaved node. If it is, the node will be added to the misbehaving list maintained by PathManager. Otherwise it is removed from the misbehaving list. Whatever the result of the deviation test is, trust rating of the node which publishes the information will be updated.

5.3.4 Bear Grudge

Figure 5-12 illustrates the scenario of bearing grudge towards misbehaved nodes.

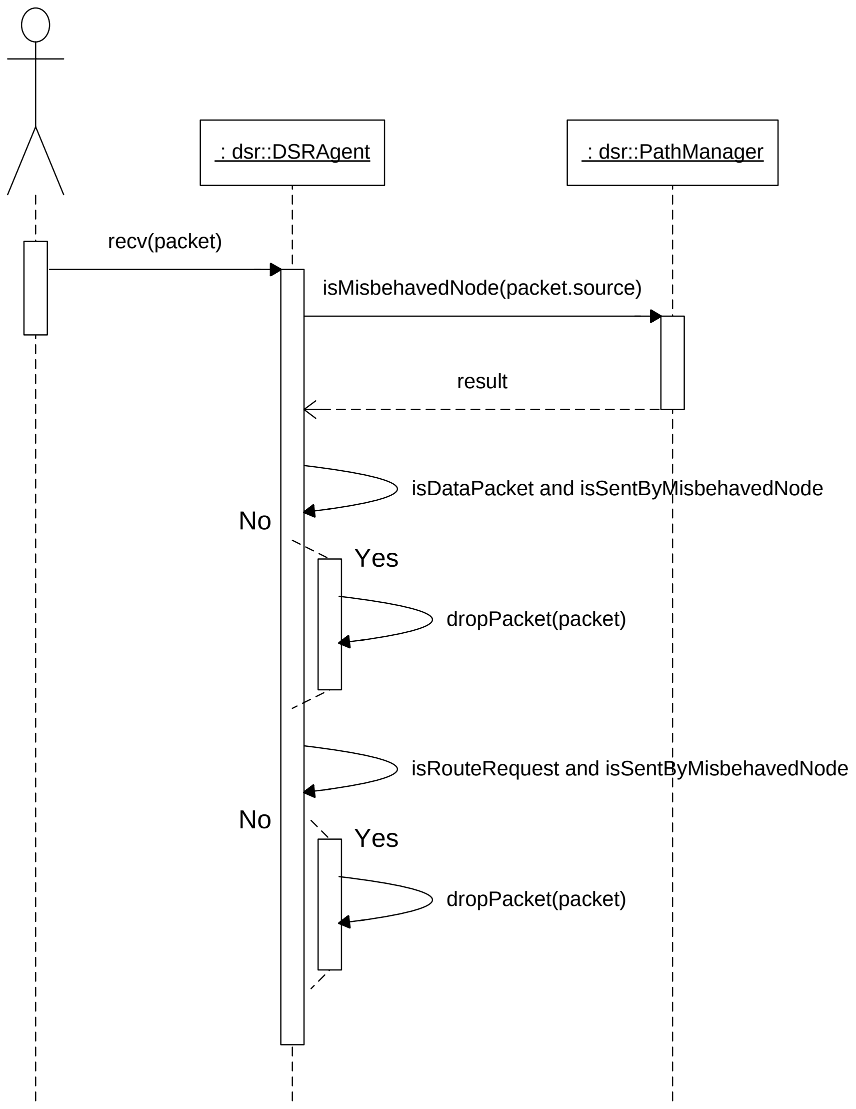

: dsr::DSRAgent
: dsr::PathManager
recv(packet)
isMisbehavedNode(packet.source)
result
isDataPacket and isSentByMisbehavedNode
No
Yes
dropPacket(packet)
isRouteRequest and isSentByMisbehavedNode
No
Yes
dropPacket(packet)

Figure 5-12 Sequence diagram of bearing grudge towards misbehaved nodes

When DSRAgent receives a packet, it queries PathManager about whether the originator of the packet is misbehaved by calling PathManager->isMisbehavedNode(packet.source). If it is misbehaved, and it satisfies the condition of the bearing grudge policy analyzed in section 4.2.3, then the packet will be dropped.

5.4 Trace and Parser

When running the simulations, some information about the mobile nodes must be logged for the purpose of performance evaluation. For example, to analyze the good throughput, we need to log how many data packets are sent from good nodes and how many are received. To analyze the misbehaved node identification rate, we need to log how many nodes are identified by each node during certain period of time and whether they are identified correctly.

We use two different ways to log the information for different modules.

- Ns-2 trace function. DSRAgent uses ns-2 trace function to log packets that are sent, received or dropped. The trace function also logs other detailed information such as the originator and destination of the packets as well as the reason of packet dropping.
- C++ iostream function. All the CONFIDANT modules use C++ iostream to log the information in text file.

As shown in Figure 5-13, DSRAgent logs the packet sending and receiving information using ns2 trace function and CONFIDANT modules logs information using C++ iostream functions. After the simulation finishes, DSRParser parses the log files and get statistic data.

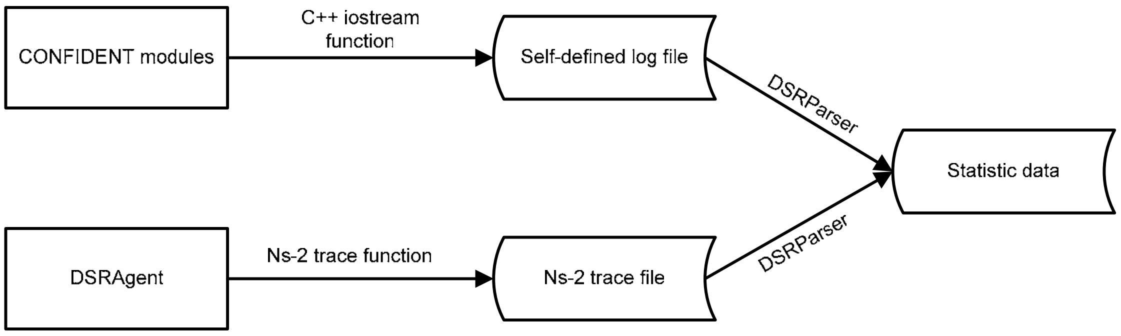


Figure 5-13 Trace and parser

The reason why ns2 trace function is not used by CONFIDANT modules is that to use the ns2 trace function, the class must be extended from the Agent class which is not necessary for CONFIDANT related classes.

All the trace files are normal text files and can be parsed to give statistic results. DSRParser is designed to parse the file and Figure 5-14 shows the class diagram.

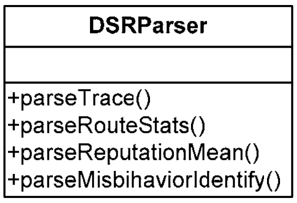


Figure 5-14 Class diagram of DSRParser

The methods of DSRParser are explained as follows.

parseTrace() – the method is used to parse the DSR trace file.

parseRouteStats – the method is used to parse the trace file of the cached routes.

parseReputationMean() – the method is used to parse the trace file of reputation mean value of nodes.

parseMisbehaviorIdentify() – the method is used to parse the trace file of identified misbehaved nodes.

5.5 Summary

This chapter gives the design of the software from the high level architecture to the detailed class design of the modules. The design covers following modules:

- The Monitor module handles attack detection and reports firsthand information.
- The Reputation System module maintains and updates the reputation rating based on the reported firsthand information and secondhand information. It also distributes the firsthand information periodically.
- The Trus tManager module maintains the trust rating and judges whether a node is trustworthy or not.
- The Path Manager module maintains the route cache, responsible for selecting and deleting routes.

Following major dynamic behaviors of CONFIDANT are described:

- Handle first hand information
- Publish first hand information
- Handle second hand information published by other nodes
- Bear grudge to misbehaved nodes

The chapter also presents the solution for trace and parser.

6 Implementation and Tests

The CONFIDANT protocol, parsers and simulations script are developed in three different computer languages of C++, Java and TCL. In this chapter, we briefly introduce the special language features and methods employed in the implementation.

6.1 Using C++ Standard Template

The three types of ratings kept by each node are dynamic. The sizes of the ratings cannot be decided at the compilation time but instead grow at run time. The rating of a node is indexed and searched by its network ID. To support the dynamic, scalable and quickly searchable features of ratings, the C++ Standard Template Library (STL) such as map, list and iterator are used. Following pseudo code give an example of how to use map to represent firsthand information.

```
map<nsaddr_t, Rating*> firsthandinfo_t;   //define the firsthand
information table
map<nsaddr_t, Rating*>::iterator it;      //create an iterator
it = firsthandinfo_t.find(address);       //search the rating of a node
if (it != firsthandinfo_t.end())          //If the rating is found
{
  (it->second)->updateRating(alpha, 1-alpha);    //update the rating
}
```

As seen in the above code, the firsthand information rating is defined as STL map class. Each element in the map contains a (index, rating) pair. When we want to search the rating of a node, the STL iterator class is used to find the rating corresponding to the given network ID.

6.2 Implementing Simulation Script

The ns2 simulator is started with an Otcl script which configures all the simulator related parameters such as the number of nodes, the routing protocol, the size of the simulated network area, etc. The script is implemented by tailoring a template for the general mobile ad hoc network. Please refer to Appendix K.10 for the source code.

6.3 Simulation Batch File

In our performance analysis, there are large number of simulations need to be done. Take the example of the simulations for evaluating throughputs of CONFIDANT, we need to test 5 different level percentages of evil nodes on 5 different scenarios each. If each simulation takes about half an hour, then it takes $5 \times 5 \times 0.5h$ = 12.5 hours for the simulations. It would be a tedious job if the developer monitors the computer for 12.5 hours and manually set up for each simulation. A better way is to implement a batch file so that the simulations can be executed automatically even during night. For this purpose shell batch files are developed which can be seen in the Appendix K.11 and K.12.

6.4 Test

We conducted following functional test for the system.

- The publish firsthand information are sent periodically and received correctly.
- Forwarding or dropping packets are detected.
- Firsthand information ratings, reputation ratings and trust ratings are updated in the correct way.
- The evil nodes drop certain types of packets.
- Misbehaved nodes are identified and routes containing these nodes are avoided.
- Normal nodes bear grudge to the misbehaved nodes.

The output can be generated for verification when the CONFIDANTVERBOSE and CONFIDANTDEBUG are set on in the hdr_confidant.h file.

The tests have been conducted with a simple network. For example, only six nodes are presented in a network with range of 250m × 250m. The number of packets sent is limited so that they can be easily tracked.

6.5 Summary

In this chapter we describe detailed implementation methods and language features by illustrating examples. We also discussed the functional test cases and the test context we used in testing. For detailed source code and configuration files please refer to Appendix K.

7 Performance Analysis

This chapter evaluates the performance of the CONFIDANT protocol. We first design the metrics for the performance evaluation. Then we introduce the important factors related to the network simulator, DSR and CONFIDANT. We conduct preliminary simulations to optimize the primary factors of CONFIDANT. After that simulations with different percentages of the evil nodes are conducted and the results are compared with that of standard DSR. Finally we analyze the performance of some variations of CONFIDANT.

7.1 Metrics

<u>**Throughputs and evil drop rate**</u>

Throughput is the most important metrics in our performance evaluation. Since the purpose of CONFIDANT is to improve the throughput for good nodes while bearing grudges to evil nodes, we evaluate the throughputs of good nodes and evil nodes separately. For simplicity, we call them ***good throughput*** and ***evil throughput***. The goal of the CONFIDANT protocol is to increase the good throughput while decrease the evil throughput as much as possible.

The formula used to calculate the good throughput is expressed in Equation 7-1.

$$GT = \frac{\sum_{i=1}^{n} \text{Received good packets}}{\sum_{i=1}^{n} \text{All good packets}}$$ **Equation 7-1**

where good packets are the packets orginiated by good nodes.

The formula for evil throughput is expressed in Equation 7-2.

$$ET = \frac{\sum_{i=1}^{n} \text{Received evil packets}}{\sum_{i=1}^{n} \text{All evil packets}}$$ **Equation 7-2**

where evil packets are the packets orginiated by evil nodes.

We also use evil drop rate to evaluate how effective CONFIDANT is to mitigate packet drop attack. Evil drop rate means how much packets are dropped by evil nodes compared to the total number of packet dropped. Equation 7-3 is used to calculate the evil drop rate.

$$EDR = \frac{\sum_{i=1}^{n} \text{Packets dropped by evil nodes}}{\sum_{i=1}^{n} \text{Total dropped packets}}$$ **Equation 7-3**

Overhead

CONFIDANT introduces network overhead by publishing firsthand information periodically. The quantity of the overhead depends on the publish timeout. CONFIDANT may also increase Route Request message since it uses stricter route selection strategy.

There are two factors in calculating overhead, the number of packets and the size of an individual packet. These two values cannot be simply multiplied since sending off a packet and transmitting a packet have different cost. Due to the time limitation we only consider the number of packets in our evaluation.

Misbehavior identification rate

In this performance analysis, we evaluate how much percentage of misbehaved nodes are identified in the network. The rate reflects how effective CONFIDANT is in identifying misbehaved nodes. To evaluate the identification rate, the average number of identified misbehaved nodes should be calculated periodically.

False positive and negative

As analyzed in section 4.2.2.4, sometimes a good node will be considered as misbehaving. We call this ***false positive***. On the other hand, some evil nodes cannot be identified and are considered as normal nodes. This is called ***false negative***. The false positive rate and false negative rate reflect how effective CONFIDANT is in identifying misbehaved nodes.

7.2 Simulation strategy

To form the comparison, simulations are conducted for Standard DSR, CONFIDANT and other modified versions of CONFIDANT. The outcomes of the simulations are compared to see whether the CONFIDANT have improvement or deterioration over the standard DSR in perspectives of the metrics, and whether the variations of CONFIDANT has different impact on the network performance.

The most important factors that impact the simulation results are the topology and traffic connections of the network. To dynamically simulate the network, following files are generated randomly and automatically to get different topology and traffic pattern.

- Node-movement scenario – specify how mobile nodes move in the network. E.g. the number of nodes participating in the network, the position of each node, the maximum speed of the movement.

- Traffic pattern – specify how a node sends packets to another. E.g, the packet sent rate, the application protocol and the size of packet.

Ns2 provides automation tools to generate the different node-movement scenario files and the traffic pattern files. The commands that are used to generate the files can be found in Appendix H. Once the files are generated, the topology and the traffic connections are deterministic. That means, the results are the same for any simulations that use the same pair of node-movement scenario and traffic pattern files.

To ensure that the simulation results are not gained by using an exaggerated positive scenario, five different node-movement scenarios are used to do the simulations in consecutive for each comparison. The result is obtained by calculating the mean of five simulation results.

For each simulation, a certain percentage of the nodes participating in the network are simulated as evil nodes. To form the comparison, these evil nodes are pre-defined instead of randomly selected.

7.3 Simulation of Evil Nodes

Most of the simulations in this project are conducted with 50 mobile nodes in the network. Among the 50 nodes, a minimum of 0% and maximum of 80% evil nodes are used. In the simulations, every node is associated with a unique network ID which ranges from 0 to 49. The evil nodes are simulated in this way. 40 out of the 50 nodes are pre-defined as candidate evil nodes. The network IDs of these 40 nodes are not contingent or sequential. When a certain number of evil nodes are supposed to be present in the network, for example 20 nodes, the first 20 nodes out of the 40 candidates are selected to act as evil nodes. The rest of the nodes behave normally even they are candidate evil nodes.

Although there are various types of attacks on DSR, we concentrate on simulating packet drop attack for the purpose of performance evaluation because its impact on the network performance can be measured directly. The types of packets that are dropped by evil nodes are specified in section 4.4.2.

7.4 Parameters

As discussed in section 2.3.1, there are two categories of parameters in the simulations, factors and primary factors. Factors can be determined through theoretic analysis or experience while primary factors have to be tested and tuned during simulations. This section describes the factors and their values we have chosen. In the next section we will explain how primary factors are decided by analyzing simulation results.

7.4.1 Ns2 Related Parameters

Table 7-1 lists the most important ns2 related parameters and the values that will be used in the simulations. Most of the values are the same with those Sonja used in her thesis [7] so that the results of simulations can be compared. However the value of transmission rate, packet size and maximum speed are different. The reason why these parameters are chosen differently is explained in the section 4.3.1.

Parameter	Value
Application traffic	CBR
Radio Range	250 m
Packet Size	64 bytes
Transmission rate	2 packets/s
Pause time for nodes	100 s
Maximum speed	1 m/s
Simulation time	900 s
Number of nodes	50
Data connections	30
Area	1000 m × 1000 m
Available bandwidth	2 Mbps

Table 7-1 Ns-2 related parameters

As seen in the table, Constant Bit Rate (CBR) is chosen as the application protocol because it is simple and it makes the results easier to analyze. 50 nodes are present in the area of 1000×1000 m^2. Among the 50 nodes, 30 nodes are connected to send CBR packets to one another. The simulation lasts 900s so that enough packets are sent to eliminate any deviations. Normally two network bandwidths are available for wireless network, 2 Mbps and 11 Mbps. Since the transmission rate is 2 packet/s and packet size is 64 byte, the traffic cannot exceed 2Mbps even routing overhead are counted. Thus 2 Mbps is chosen as available bandwidth.

7.4.2 DSR Related Parameters

Table 7-2 lists the most important parameters related to DSR protocol. The parameter names are the same with those used in the source code. The reasons why certain values are chosen for them have already analyzed in section 4.1.

Parameter	Description	Value
Snoop_source_routes	Flag used to indicate if source routes should be snooped.	True
Reply_only_to_first_ routereq	Flag used to indicate if a node should only reply to the first route reply it receives.	False
Send_grat_replies	Flag used to indicate if a node should send out gratuitous replies to shorten routes.	False
Reply_from_cache_on propagation	Flag used to indicate if a node should, when receiving a route request, reply with a route from its cache if possible.	False
Use_ring_search	Flag used to indicate if a node should, use ring search to discover new routes.	False
Dsragent_enable_flowstate	Flag used to indicate whether to use flow state extension feature	False

Table 7-2 DSR related parameters

7.4.3 CONFIDANT Related Parameters

Table 7-3 lists the parameters related to CONFIDANT protocol. The parameter names are the same as those used in the source code. As shown in the table, all the parameters have to be tested and determined in the simulations. The parameters in this table are very important because they will directly impact the performance of CONFIDANT and even slight variation will change the results of the evaluation. As introduced in section 2.3.1, these parameters are called primary factors. We will tune these primary factors in the following section.

Parameter	Description	Value
PACK_TIMEOUT	A time out value determines how long a node wait for a passive acknowledge package.	
INACTIVITY_FADING	The fading factor of first information and reputation value that a node stores when the time goes on.	
TRUST_FADING	The fading factor of trust value that a node stores when the time goes on.	
DEVIATION_THRESHOLD	The threshold in the deviation test	
SECONDHAND_INFO_WEIGHT	The weight when secondhand information is used to update the reputation value that a node stores.	
MISBEHAVIOR_TOLERANCE	A threshold that a node uses to determine whether another node is misbehaved.	
UNTRUST_TOLERANCE	A threshold that a node uses to	

	determine whether another node is trustworthy.	
INACTIVITY_TIMEOUT	The timeout that a node updates firsthand information and reputation due to inactivity.	
PUBLISHING_TIMEOUT	The timeout that a node publishes its firsthand information about other nodes.	
USING_TRUST	Whether use trust to determine whether accept second hand information.	

Table 7-3 CONFIDANT related parameters

7.5 Estimation of Primary Factors

As presented in the previous section, there are several very important factors of CONFIDANT. The values of these factors should be adjusted to get the best network performance. Through tuning the primary factors, we can also see how these factors impact the performance.

In the subsequent sections, several reasonable values are chosen to be tested for each factor. For each value, five simulations are executed consecutively for different scenarios and the results are the mean of the outcome of the five simulations unless otherwise specified.

7.5.1 Estimation of Misbehaved Threshold

The purpose of misbehaved threshold is to distinguish misbehaved nodes from good ones. (Equation 4-6 describes the usage of the misbehaved threshold.) So the threshold should be less than the mean reputation value of most evil nodes and higher than that of most good nodes. The estimated values for other primary factors can be seen in the Table 7-4.

Parameter	Value	Parameter	Value
Publish timeout	2 s	Inactivity fading	0.9
Deviation threshold	0.75	Secondhand information weight	0.2
Inactivity timeout	2 s	Percentage of evil nodes	40%
PACK timeout	0.5s		

Table 7-4 Parameters used when estimating misbehaved threshold

To estimate the threshold value we take several steps to analyze the simulation results and approximate the value to the best selection.

Step 1: Analyze average mean reputation value

We first conduct a simulation to analyze the mean reputation value of each node. This step has following purposes.

1) Prove that there exists a misbehaved threshold that can distinguish misbehaved nodes from normal nodes. It requires that the intervals of the mean reputation values of misbehaved nodes and normal nodes do not overlap.
2) Select the range of misbehaved threshold so that the reasonable values will be tested to get the best simulation result.

In CONFIDANT, each node keeps reputation rating about any other nodes that it has communicated or heard about in the network. The ***mean reputation value*** is calculated according to Equation 2-4 and it indicates whether a node misbehaves or not when compared with misbehaved threshold. If the mean reputation value of a node is greater than the misbehaved threshold, it is considered as misbehaved node. Otherwise it is considered as normal node.

It is meaningless if we look at the mean reputation value of only a few nodes due to the deviation. A better way is to analyze the average mean reputation value of any node stored by all other 49 nodes. As discussed in section 2.3.2, there are three alternative methods to calculate the average of sample data. Which one to use depends on the nature of data set and what is of interest to the user. Before analyzing the data we don't know which one should be used. However if the reputation system works in a correct way, the results of the three methods should look similar. Thus we calculate the average using all three methods and compare them.

The results of the simulation are shown in Figure 7-1. Because of the space limitation, here we only present the average mean reputation values of the first 20 nodes. The results for all the 50 nodes can be seen in the Appendix F. As seen in the figures, the mean, median and mode of the mean reputation values look very similar. That means any one of the three kinds of values is meaningful to be used as the average. The modes for a few nodes are missing in the figure because there are several possible values for each of the modes and the word processing tool just doesn't know how to display them. But after checking the data source, those values are very similar to mean or median of the same node.

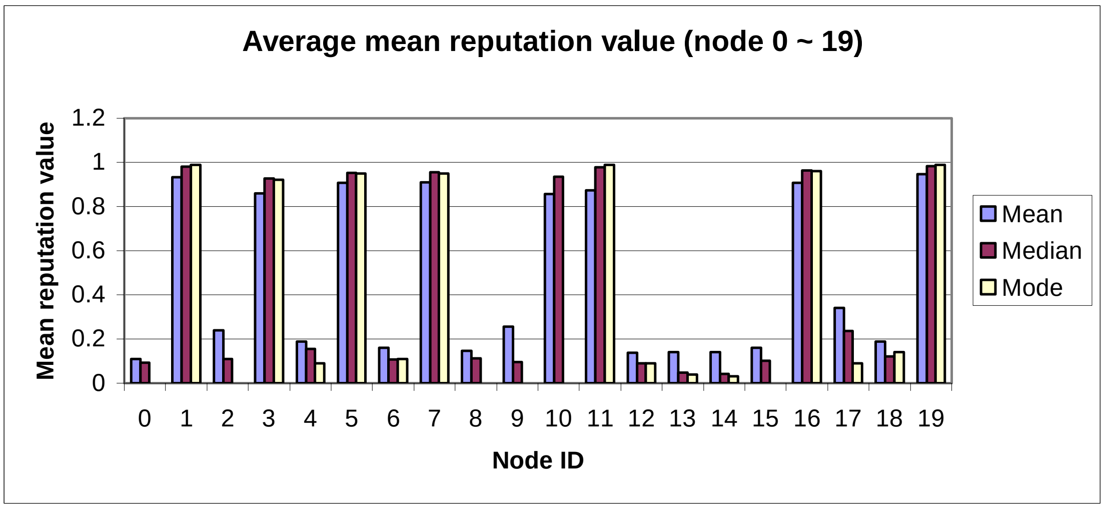


Figure 7-1 Average mean reputation value of mobile nodes in the network

As seen in the figure, the average mean reputation values of the misbehaved nodes are mostly higher than 0.8 while the values of good nodes are lower than 0.5. In our simulation the actual evil nodes in the simulation are 1, 3, 5, 7, 10, 11, 16, 19, etc. The figure shows that the mean reputation values higher than 0.8 absolutely match the actual evil nodes. This result proves the purpose 1) and also provides basis for choosing an appropriate range to tune the misbehaved threshold further. After analyzing average mean reputation values for more scenarios, we have found that the misbehaved threshold should be a value greater than 0.8.

Step 2: Adjust misbehaved threshold

Having estimated a gross range, we conduct more simulations to choose the best misbehaved threshold. Figure 7-2 shows the good throughput and evil throughput at different misbehaved threshold. We can see that the lower the threshold, the lower both the good and evil throughputs. Thresholds 0.85 and 0.9 are more favorable because the evil throughputs are very low while good throughputs are medium.

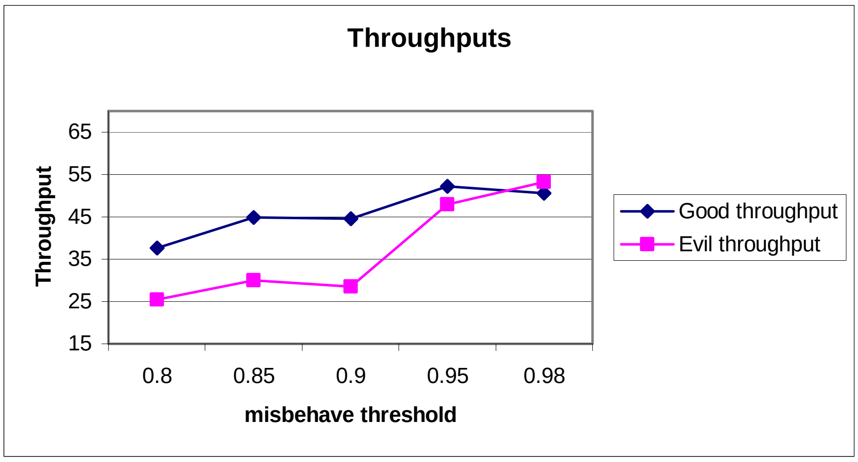


Figure 7-2 Throughputs with different misbehaved threshold

Step 3: Analyze the misbehaved identification rate and false negative rate

Now we have narrowed our selection within two values, 0.85 and 0.9. We further analyze the misbehaved identification rate and false negative rate to decide which one to choose.

Figure 7-3 shows the misbehaved identification rate for different misbehaved threshold. Figure 7-4 shows the false negative rate at different time. We can see that the identification rate of threshold 0.9 is slightly higher than that of threshold 0.85. However, in Figure 7-4, the threshold 0.9 has lower false negative rate. Thus we choose 0.9 as misbehaved threshold.

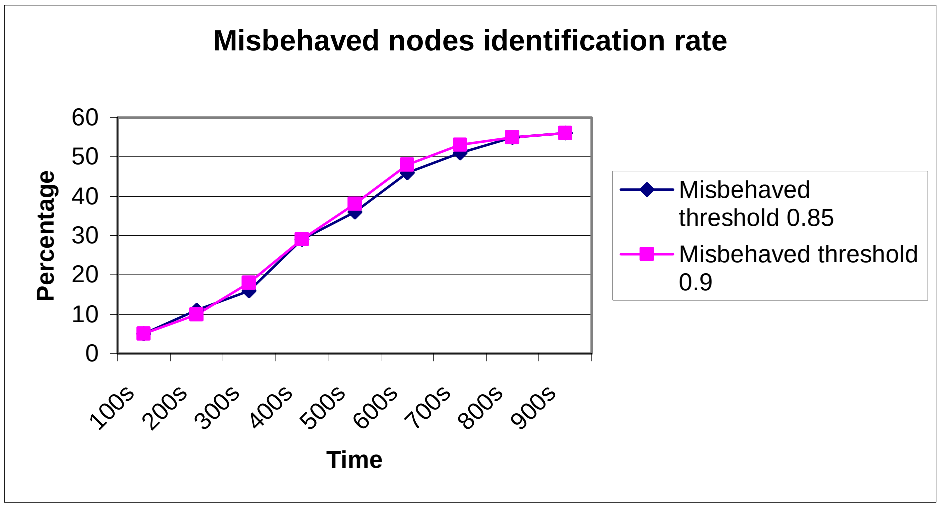


Figure 7-3 Misbehaved nodes identification rate

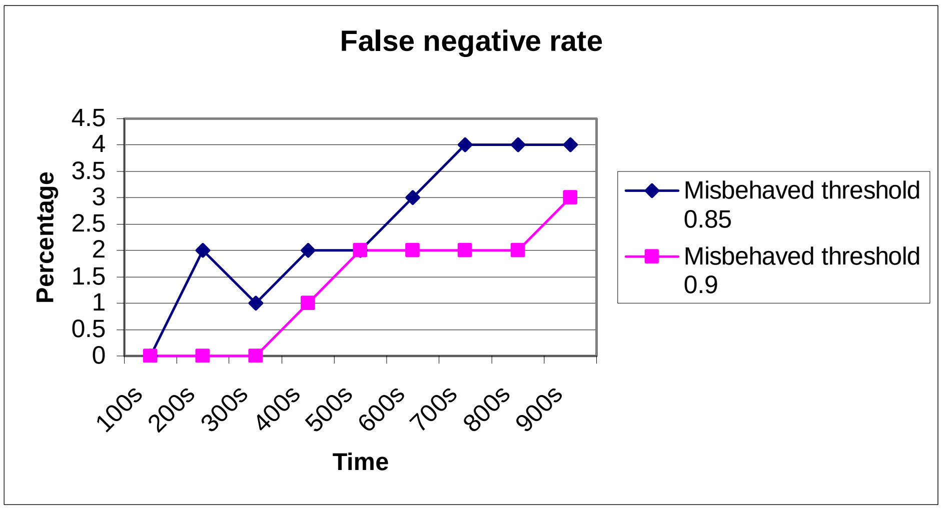


Figure 7-4 False negative rate

Summary: Through the three-step analysis, we get a misbehaved threshold which results in low evil throughput, medium good throughput and precise identification of misbehaved nodes. Through the analysis we can see that sometimes throughput is not the only criteria to select the value for a parameter. We should also consider the purpose of the parameter and analyze other results that are directly affected by the parameter.

7.5.2 Estimation of Deviation Threshold

The deviation threshold is used in deviation test to decide whether to accept second hand information or not. If the difference between the second hand information and the reputation rating about a certain node is greater than the deviation threshold, the second hand information will be discarded. Otherwise it will be accepted and updated into the reputation rating. Table 7-5 shows the factors and their values we used to do the simulations.

Parameter	Value	Parameter	Value
Publish timeout	2 s	Inactivity fading	0.9
Misbehaved threshold	0.9	Secondhand information weight	0.2
Inactivity timeout	2 s	Percentage of evil nodes	40%
PACK timeout	0.5s		

Table 7-5 Factors used when estimating deviation threshold

The simulation results are presented in Figure 7-5. As seen, the highest good throughput is about 46% when the deviation threshold is 0.5. Although the evil throughput at that time is about 27% which is not the lowest, it is the point where high good throughput and relatively low evil throughput are achieved. Thus we choose 0.5 as the deviation threshold.

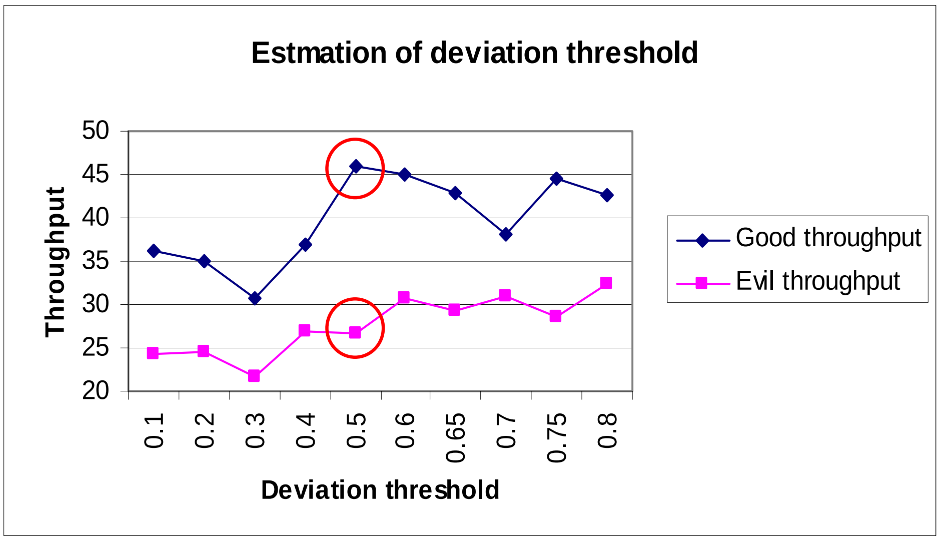


Figure 7-5 Estimation of deviation threshold

7.5.3 Estimation of Publish Timeout

The publish timeout determines how often a node publishes its own first hand information about other nodes. Since a node sends packets every 0.5 s in our simulation, it is supposed to update firsthand experience about other nodes every 0.5 s. Thus the publish timeout should be at least 0.5 s. The more frequent a node publishes information, the more precisely the other nodes know about their environment. In this sense, it is better for a node to publish information frequently. However, more publish packets means more network overhead which could deteriorate the network performance. So the publish timeout need to be balanced. Table 7-6 shows the factors and their values we used to do the simulations.

Parameter	Value	Parameter	Value
Deviation threshold	0.5	Inactivity fading	0.9
Misbehaved threshold	0.9	Secondhand information weight	0.2
Inactivity timeout	2 s	Percentage of evil nodes	40%
PACK timeout	0.5s		

Table 7-6 Parameters used when estimating publish timeout

The results of the simulation are presented in Figure 7-6. As seen, the highest good throughput is about 48% when the publish timeout is 3 s. Although the evil throughput of other publish timeout, e.g. 0.5 s and 1 s, are also on the similar level, 3 s is more reasonable because its evil throughput is much lower. Sometimes it is hard to choose between 2 s and 3 s because their throughputs are similar. However, publish timeout 2 s introduces more network overhead. Thus we choose 3 s as the publish timeout.

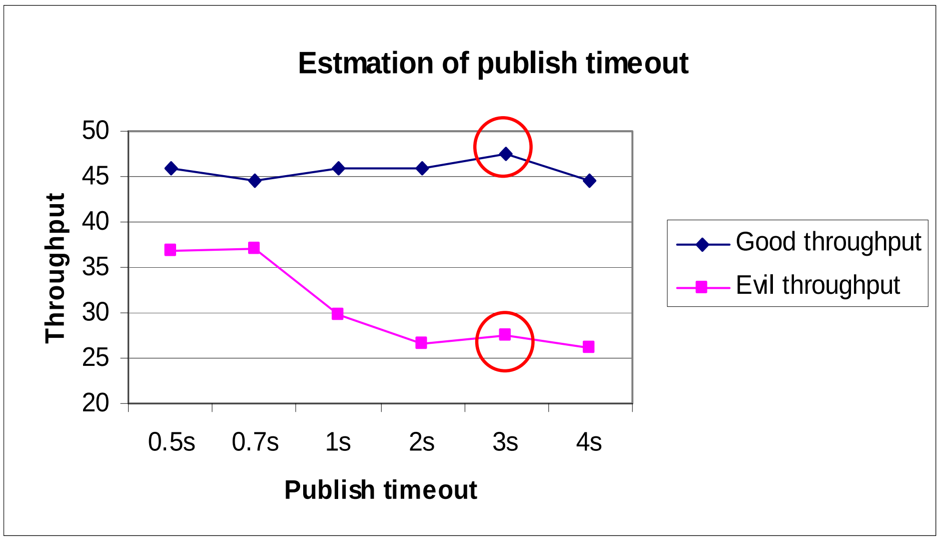


Figure 7-6 Estimation of publish timeout

7.5.4 Estimation of Secondhand Information Weight

The secondhand information weight is used to decide how much second hand information is taken to update the reputation rating. The higher the weight, the more the reputation rating is influenced by recommendation. Table 7-7 shows the other factors and their values we used to do the simulations.

Parameter	Value	Parameter	Value
Deviation threshold	0.5	Inactivity fading	0.9
Misbehaved threshold	0.9	Publish timeout	3 s
Inactivity timeout	2 s	Percentage of evil nodes	40%
PACK timeout	0.5 s		

Table 7-7 Factors used when estimating secondhand information weight

The results of the simulation are presented in Figure 7-7. As seen, the network gets the highest good throughput about 47% when the secondhand information weight is 0.2 or 0.4. However, the throughput for evil node at 0.2 is lower than that of 0.4. Thus we choose 0.2 as optimized secondhand information weight.

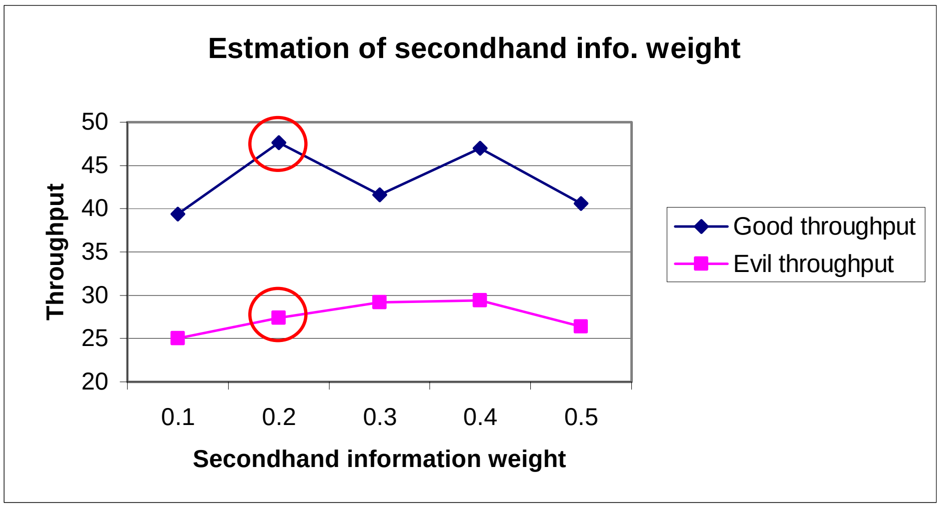


Figure 7-7 Estimation of secondhand information weight

7.5.5 Estimation of PACK Timeout

PACK timeout is used for a node to judge whether its next hop forwards the packet successfully or not. If no PACK packet is received during the timeout, the next hop is considered misbehaved. Usually PACK timeout should be less than 0.5 second in our simulation because the data packets are sent at about every 0.5 second and it would be more effective to let a node know whether its next hop misbehaved before the node sends the following packets. However, since in real implementation each node sends data packets at (0.5 + random) seconds, we also test the PACK timeout a little bit greater than 0.5 seconds. Table 7-8 shows the other factors and their values we used to do the simulations.

Parameter	Value	Parameter	Value
Deviation threshold	0.5	Inactivity fading	0.9
Misbehaved threshold	0.9	Publish timeout	3 s
Inactivity timeout	2 s	Percentage of evil nodes	40%
Secondhand info. weight	0.2		

Table 7-8 Factors used when estimating PACK timeout

The results of the simulation are presented in Figure 7-8. As seen, when the PACK timeout is 0.5 s, we get relatively high throughput which is about 48% and relatively low throughput which is about 27%. However, we can see that the throughputs of PACK timeout 0.7 s also look good. Actually, more simulations show that PACK timeout 0.5 s and 0.7 s have similar impact on the throughput. Either 0.5 s or 0.7 s can be chosen. We pick 0.5 as PACK timeout.

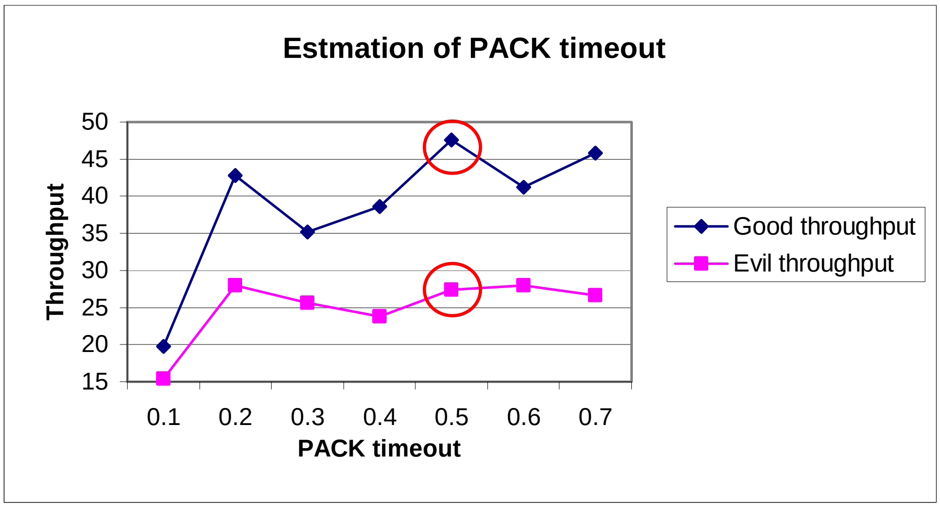


Figure 7-8 Estimation of PACK timeout

7.5.6 Estimation of Inactivity Timeout and Fading Factor

Inactivity timeout and fading factor work together to make discount to the past experience and put more emphasis on the recent experience about another node. The two parameters are so interrelated that they should be considered at the same time. Take the fading of reputation rating as an example, when the inactivity timeout decreases or the fading factor increases, the reputation rating fades faster. That means, decreasing inactivity timeout and increasing fading factor have similar impact on the throughputs. Thus, we only adjust inactivity timeout to optimize the performance. Table 7-9 shows the factors and their values we used to do the simulations.

Parameter	Value	Parameter	Value
Deviation threshold	0.5	Inactivity fading	0.9
Misbehaved threshold	0.9	Publish timeout	3 s
Secondhand info. weight	0.2	Percentage of evil nodes	40%
PACK timeout	0.5 s		

Table 7-9 Factors used when estimating inactivity timeout

The results of the simulation are presented in Figure 7-9. As seen, when inactivity timeout is 2 second, we can get relatively high good throughput which is about %48 and low evil throughput which is about 27%. Thus we choose 2 seconds as inactivity timeout.

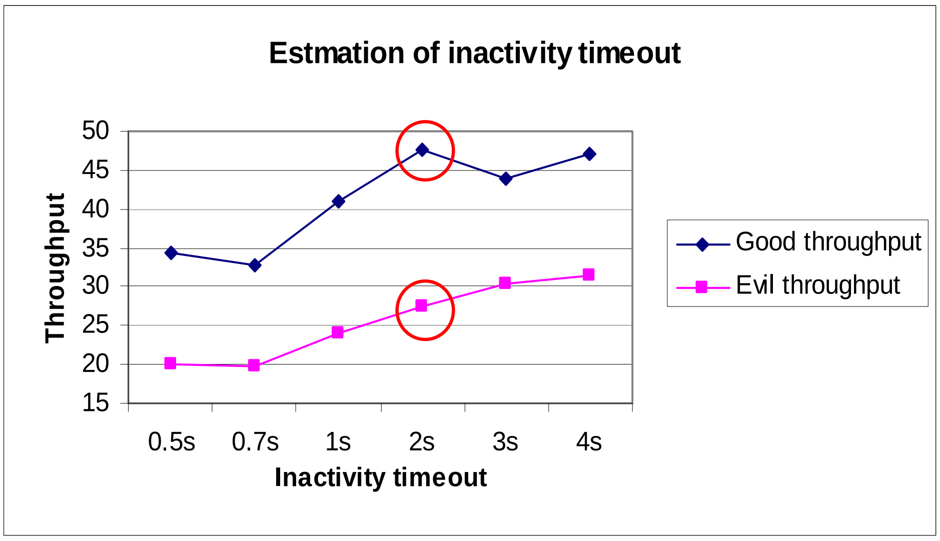


Figure 7-9 Estimation of inactivity timeout

7.5.7 Estimation of Trust Threshold

Trust threshold is used to judge whether a node is trustworthy or not as a recommender in the network. If a node is trustworthy then secondhand information published by that node will be accept. Otherwise the information will be discarded. Table 7-10 shows the other factors and their values we used to do the simulations.

Parameter	Value	Parameter	Value
Deviation threshold	0.5	Inactivity fading	0.9
Misbehaved threshold	0.9	Publish timeout	3 s
Secondhand info. weight	0.2	Percentage of evil nodes	40%
PACK timeout	0.5 s	Inactivity timeout	2 s

Table 7-10 Factors used when estimating thrust threshold

The results of the simulation are presented in Figure 7-10. As seen, the good throughputs for all four thresholds are almost same. However, lower evil throughput is achieved when the threshold is 0.95. Thus we choose 0.95 as trust threshold.

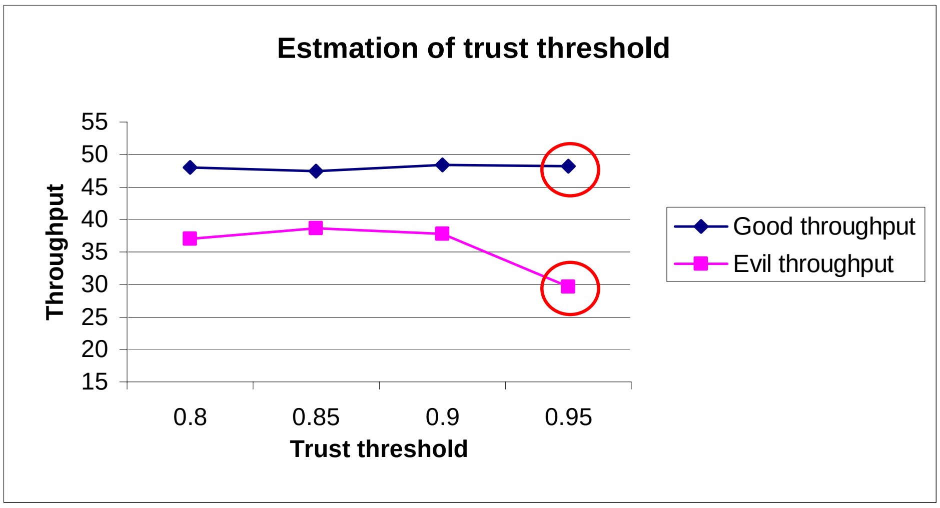


Figure 7-10 Estimation of trust threshold

7.6 Performance Evaluation

In this section, we will evaluate the performance of CONFIDANT protocol when different percentages of evil nodes are present in the network. Since we have estimated the values of primary factors in the previous section, the values in Table 7-10 except percentage of evil nodes will be used to do all the simulations in this section.

Apart from evaluating the performance of the classic CONFIDANT protocol proposed in [7], we will also analyze two variations of CONFIDANT, Path Re-ranking and Using Trust, to see how they will impact the performance of the network and whether they have advantages over the classic CONFIDANT. During the evaluation, we use following name conventions for different variations of CONFIDANT protocol.

- Standard DSR – the DSR protocol specified in [1].
- CONFIDANT – the classic CONFIDANT protocol specified in [7].
- Path Re-ranking – the CONFIDANT protocol with path re-ranking as route selection strategy.
- Using trust – the CONFIDANT protocol which uses trust value to decide whether to accept second hand information.

7.6.1 Throughputs and Evil Drop Rate

We conducted the simulation with different percentages of evil nodes 0, 20%, 40%, 60% and 80%. The results are compared with those of standard DSR when the same percentages of evil nodes are present in the network.

7.6.1.1 Good Throughput

Figure 7-11 presents the good throughputs of CONFIDANT and standard DSR. It is surprised to see that the good throughput of CONFIDANT does not show any improvement over that of standard DSR, whereas according to [7] CONFIDANT can improve good throughput by two times. Our result does not fulfill the goal of CONFIDANT that the throughput of the network should be increased by discouraging misbehavior.

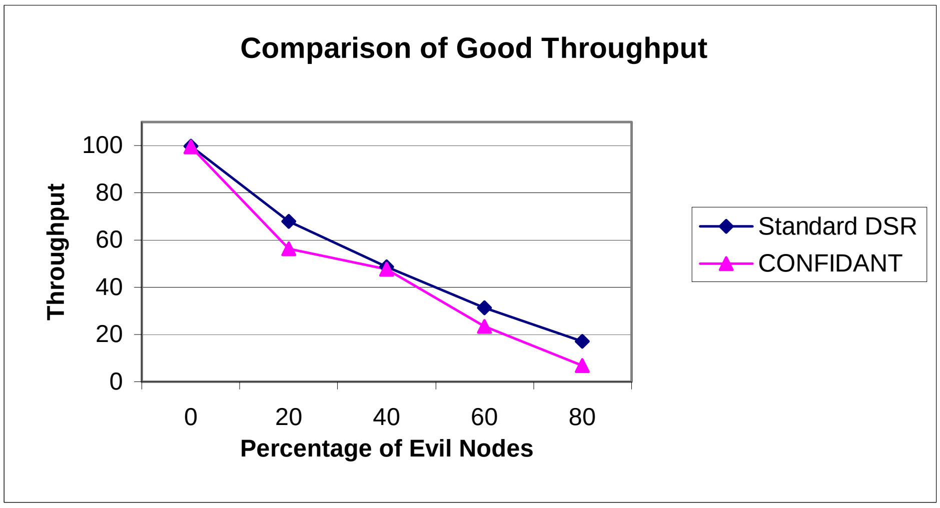


Figure 7-11 Comparison of good throughput

In section 7.6.1.4 we will further investigate the reasons why good throughput is not improved in our simulations.

7.6.1.2 Evil Throughput

Figure 7-12 presents the evil throughputs of CONFIDANT and standard DSR. As seen, the evil throughput of CONFIDANT significantly decreases up to 50% compared to that of standard DSR. This result fulfills the goal of CONFIDANT that the evil throughput should be suppressed. In the figure, the throughput at zero percentage of evil nodes is empty because we cannot calculate the evil throughput when there are no evil nodes in the network.

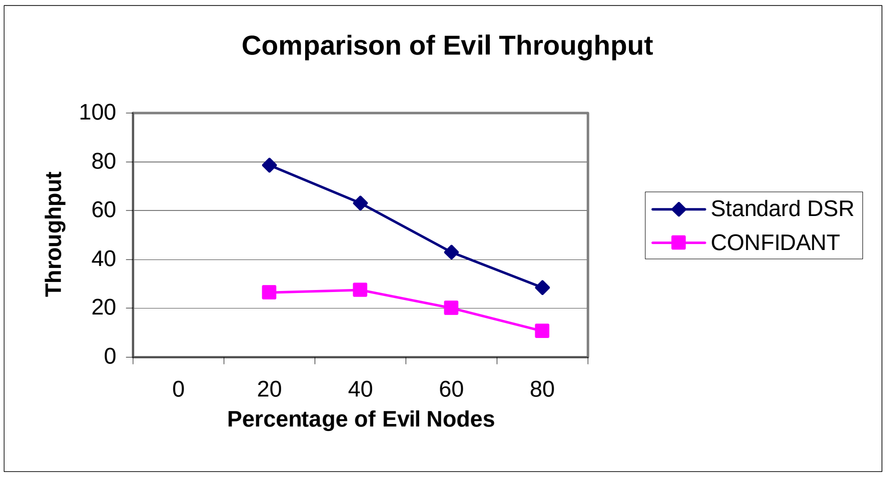


Figure 7-12 Comparison of evil throughput

The figures in Appendix G show the confidence intervals of CONFIDANT and Standard DSR regarding good throughput and evil throughput. The comparison confirms that CONFIDANT is more effective in decreasing evil throughput than Standard DSR is.

7.6.1.3 Evil Drop Rate

The evil drop rate is the percentage of packets dropped by evil nodes in all the dropped packets. It reflects how much the evil drop contributes in overall packet drop compared to other drop reasons. Equation 7-3 is used to calculate the evil drop rate. Figure 7-13 shows the evil drop rates of CONFIDANT and standard DSR. As seen in the figure, the evil drop rate of CONFIDANT is significantly lower than that of standard DSR for all percentages of evil nodes. It is kept under 6%. This means that fewer packets are routed by evil nodes. This result fulfills the purpose of CONFIDANT that the misbehaved nodes should be avoided in forwarding packets.

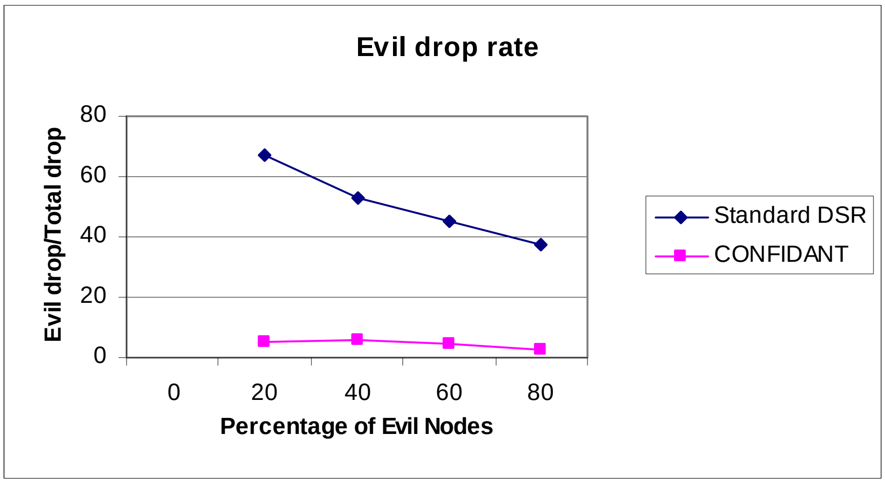


Figure 7-13 Comparison of evil drop rate

7.6.1.4 Investigate the Reason of Low Good Throughput

Since the evil drop rate is reduced significantly for CONFIDANT, why do we still get lower good throughput? We take following steps to investigate the reason.

<u>Step 1: analyze other packet drop reasons</u>

The first reason we can think of is that there may be other packet drop reasons that overturn the effect of evil drop. Figure 7-14 shows the major packet drop reasons of CONFIDANT and their percentage in the total packet drop. The result is based on the 40% of evil nodes. In the figure "drop evil source" indicates the packets dropped due to bearing grudge to evil nodes. "Send buff drop" indicates packets dropped in the send buffer of DSR. "Terminate" indicates the packet drop due to the termination of simulation.

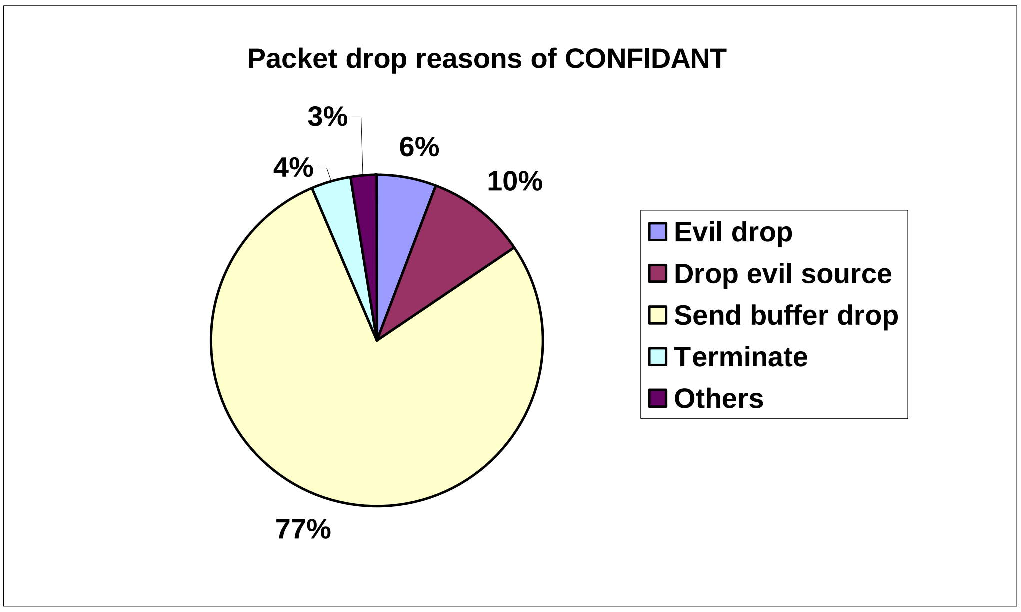


Figure 7-14 Packet drop reasons of CONFIDANT

As seen in the figure, the most influential packet drop reason is send buffer drop, which counts 77% of the total drop, whereas evil drop rate only counts 10% and has much less effect in the whole packet drop.

Figure 7-15 shows the send buffer drop rate comparison of CONFIDANT and standard DSR. As seen in the figure, CONFIDANT has generally much higher send buffer drop rate than standard DSR. The high send buffer drop rate cancels out the low evil drop rate so that the overall good throughput remains low.

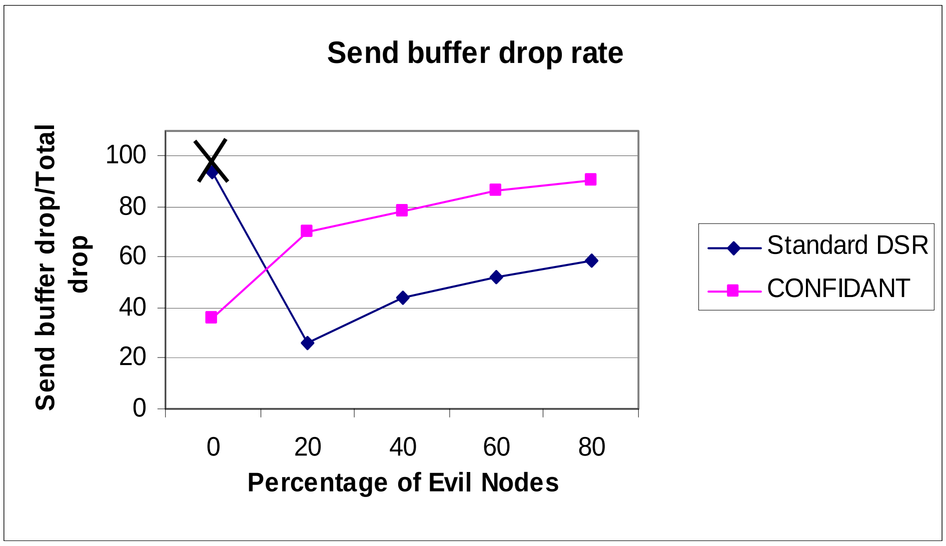


Figure 7-15 Comparison of send buffer drop rate

Note: The rate of Standard DSR at zero evil nodes should be ignored since it is abnormal. The reason why it is exceptionally high is that when there are no evil nodes present in the network, other types of packet drop are almost zero and the send buffer drop is comparatively large. For the similar reason, the statistics at zero percentage evil nodes will be neglected in the subsequent analysis.

Step 2: investigate the root cause of high send buffer drop rate

By far we have known that high send buffer drop rate is the main reason why the good throughput of CONFIDANT is not improved. However, send buffer drop is not the root cause. We must investigate why the send buffer drop rate increases significantly for CONFIDANT.

The send buffer of DSR works in this way. All data packets are saved in the send buffer before they're sent out. If a packet has not been sent out after certain timeout, it will be dropped. In most cases, a packet is dropped because no routes are found within the timeout.

There are three possible cases when CONFIDANT cannot find routes within timeout.

- Only bad routes exist in the route cache and they are discarded. A bad route is the route containing misbehaved nodes.
- No routes exist in the route cache at all.
- Good routes exist in the route cache but they are misjudged as the bad route and discarded.

The first two cases are related to the network topology. If most of the packets are dropped due to the first two cases, then it means there are no enough good routes in the network and CONFIDANT can do little to help. The third case is related to the CONFIDANT protocol. If a lot of packets are dropped due to the third case then we can argue that CONFIDANT should be blamed for not increasing good throughput.

Figure 7-16 illustrates the number of send buffer drop due to the first two different cases. As seen in the figure, majority of the packets (60% – 40%) are dropped in the send buffer because there are only bad routes which contain misbehaved nodes. However, about 20% – 40% of packets are dropped due to no routes available at all.

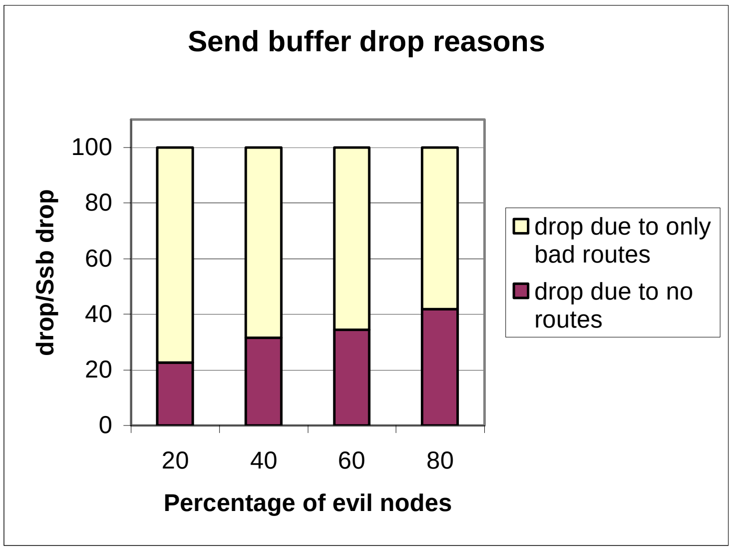


Figure 7-16 Send buffer drop reasons

Among the number of category "drop due to only bad routes" some routes identified as bad are actually good. These routes belong to third case. Figure 7-17 shows the percentage of packets that are dropped due to misjudgment in the whole send buffer drop.

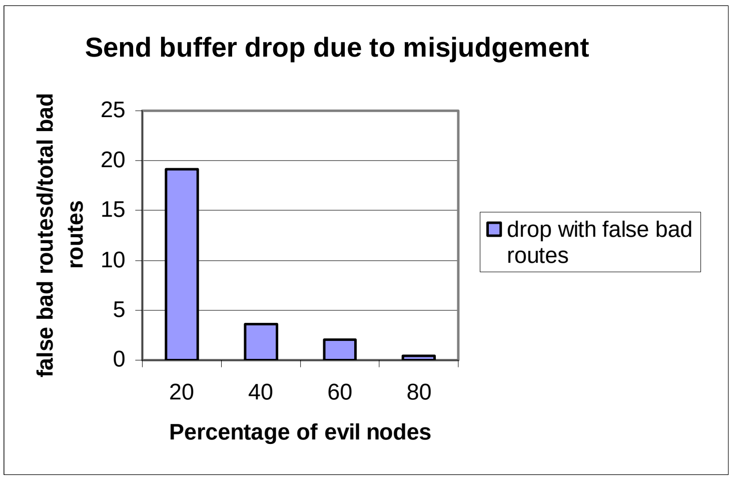


Figure 7-17 Percentage of send buffer drop due to misjudgment

As seen in the figure, the send buffer drop due to misjudgment of misbehaved nodes is very low except at the 20% evil nodes. That means in most cases CONFIDANT can correctly identify the bad routes and discard them.

So far we have seen that a great number of packets are dropped in the send buffer of DSR because no good routes can be found within the send buffer timeout. Furthermore, our test simulation shows that increasing the timeout does not help in finding good routes. Thus we think it is the problem of the simulator that there are not enough good routes available in the network.

Step 3: strengthen the conclusion

To avoid the possibility that the conclusion we got in step 2 is the result of inaccuracy of our implementation of CONFIDANT, we have devised an Ideal CONFIDANT which can 100% identify misbehaved nodes. All the behaviors of the Ideal CONFIDANT are the same with that of the classic CONFIDANT except that we let each node know which nodes are actually evil so that the evil nodes are all identified. Figure 7-18 shows the good throughput of Ideal CONFIDANT comparing to that of classic CONFIDANT and standard DSR.

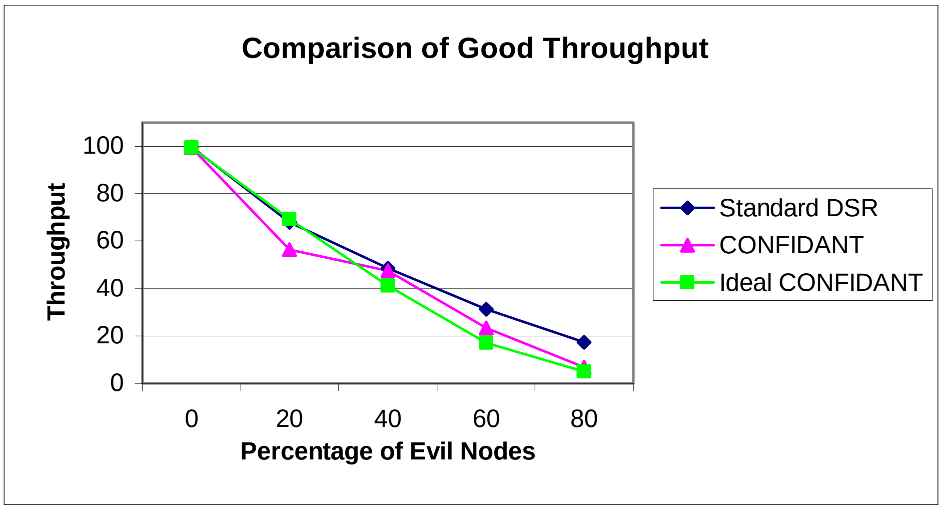


Figure 7-18 Ideal CONFIDANT

As seen, the good throughput of Ideal CONFIDANT is also lower than that of standard DSR. With the evil node percentage increasing, the good throughput of Ideal CONFIDANT decreases fast and even lower than the good throughput of Classic CONFIDANT. Majority of the packet are dropped due to the send buffer timeout because there is no enough routes available in the network.

Summary: The simulation results of classic CONFIDANT shows significant decrease in evil throughput and evil drop rate compared to standard DSR. These results fulfill the objective of the CONFIDANT. However, the good throughput of CONFIDANT does not improve as claimed in [7]. The reason is that there are no enough good routes available in the simulated network and most of the packets are dropped in the send buffer. The premise of CONFIDANT is that there are route redundant in the network. Without the route redundant CONFIDANT cannot work properly. Thus the performance of the CONFIDANT heavily depends on the simulator.

7.6.2 Overhead

As stated in section 7.1, CONFIDANT increases the network overhead by publishing firsthand information. It may also increase Route Request and Route Reply since it uses stricter route selection strategy and initiate more Route Discovery to find safe routes.

Figure 7-19 shows the network overhead of CONFIDANT and standard DSR. The overhead is divided into two categories, publish information and routing overhead. Only Route Request and Route Reply are calculated for routing overhead because we think CONFIDANT increases these two kinds of messages most. As seen, CONFIDANT significantly increases the routing overhead compared to standard DSR. The increase of routing overhead is also partly due to no enough good routes in the network. DSR keeps sending Route Request to discover new routes.

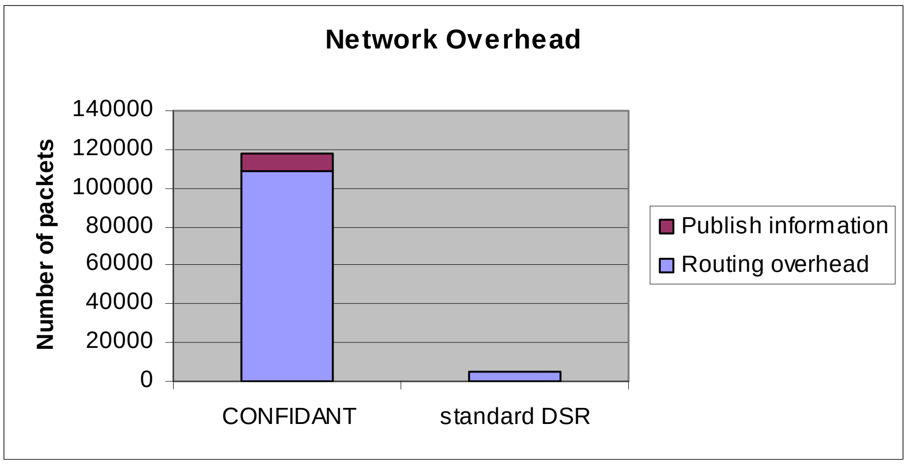


Figure 7-19 Network overhead evaluation

7.6.3 CONFIDANT with Path Re-ranking

An alternative route selection strategy is path re-ranking. With classic CONFIDANT, the Path manager only selects a route containing no misbehaved nodes, whereas with Path re-ranking the Path manager selects a route based on the reputation metrics of the route. In our implementation, we use a simple reputation metrics which is the average of the mean reputation values of all the nodes along the route. The advantage of Path re-ranking is that the packets will be sent out as long as there exists a route to the destination. Thus the send buffer drop rate would be decreased. However, Path re-ranking may have higher evil drop rate compared to the classic CONFIDANT since the selected route may contains misbehaved nodes.

The comparison of the good throughput, evil throughput and evil drop rate for path re-ranking and other protocols are shown in Figure 7-20, Figure 1-1 and Figure 7-22 separately.

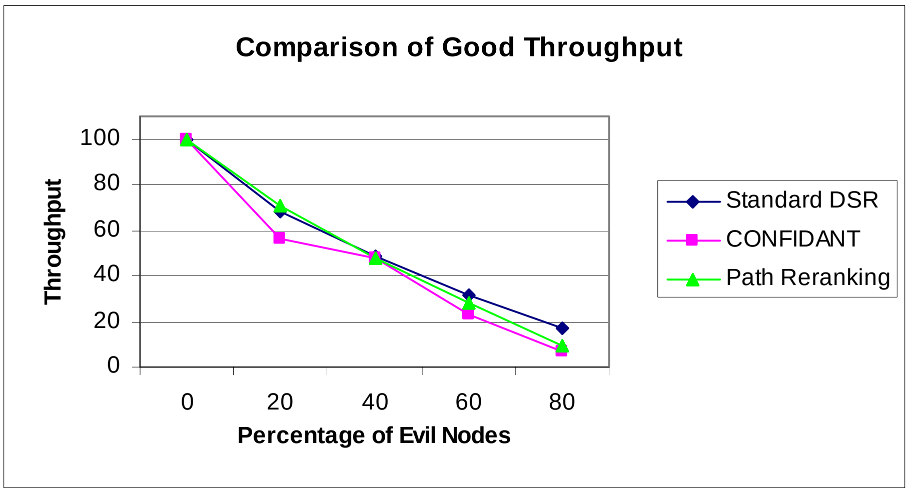


Figure 7-20 Good throughput of Path re-ranking

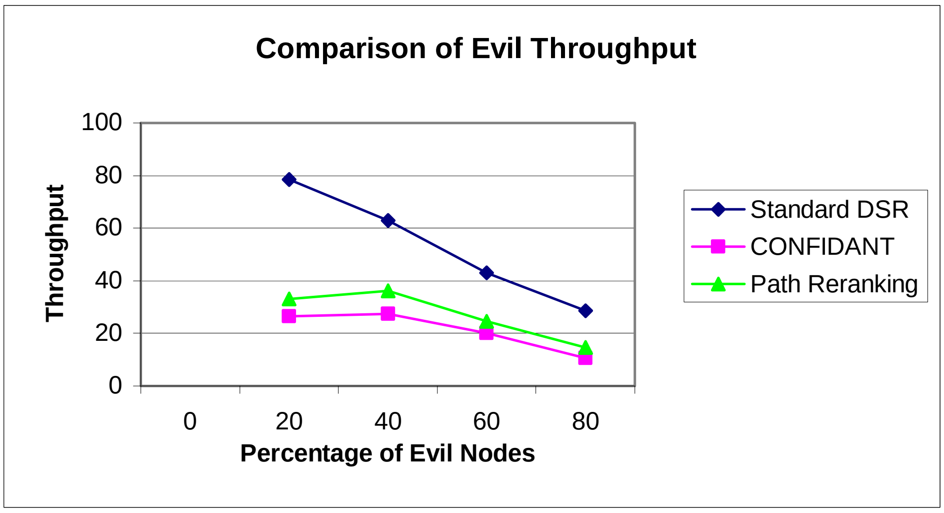


Figure 7-21 Evil throughput of Path re-ranking

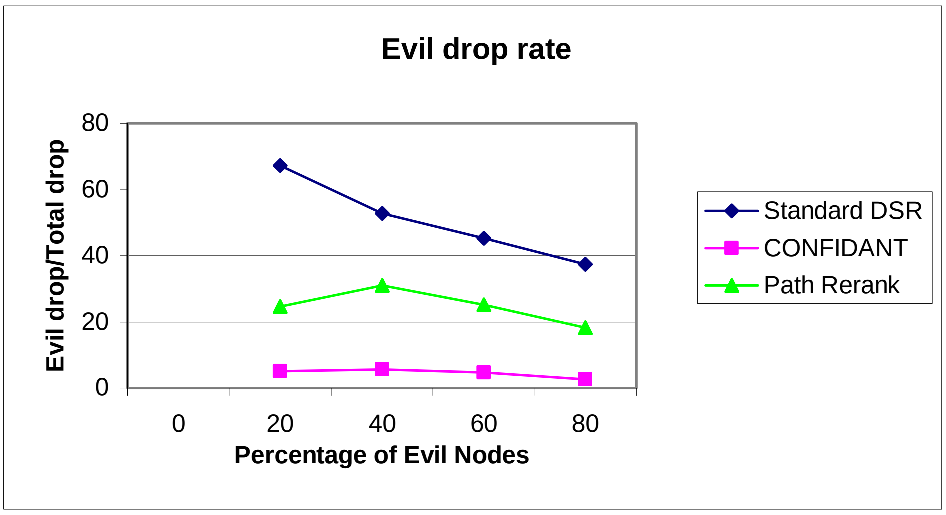


Figure 7-22 Evil drop rate of Path re-ranking

As seen in the figures, the good throughput, evil throughput and evil drop rate of Path re-ranking lie between those of standard DSR and classic CONFIDANT. Although Path re-ranking has slightly higher good throughput than classic CONFIDANT, it is not very effective at deterring evil nodes from misbehaving because the evil throughput and the evil drop rate remain high. Thus we get the conclusion that the classic CONFIDANT can better cope with misbehavior than Path re-ranking does.

7.6.4 CONFIDANT with Using Trust

Sonja [7] suggests using trust rating as an alternative way to accept the secondhand information. With classic CONFIDANT when a node receives secondhand information about other nodes, it conducts deviation test to decide whether to accept the information. The information that deviates too much from its own opinion is discarded. At the mean time, trust rating about the publisher is updated. Otherwise the information is accepted to update the reputation value.

When trust rating is used, the node accepts the secondhand information if it thinks the publisher is trustworthy, no matter whether the secondhand information deviates too much from its own opinion. Using trust can speed up detecting the misbehaved nodes because all the received secondhand information is used to influence the reputation rating directly or indirectly.

Figure 7-23 shows the misbehavior identification rate of using Trust and its comparison with classic CONFIDANT. It is surprised to see that instead of speeding up detection time, using trust actually slows down the detection than classic CONFIDANT does. Due to the time limitation, we do not investigate further why it happens. It could be a future work.

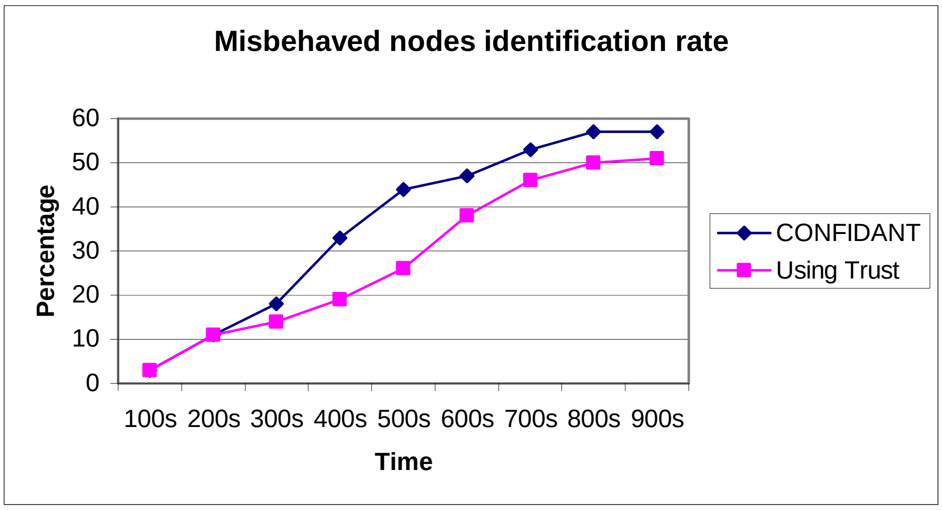


Figure 7-23 Misbehavior identification rate of Using trust

7.7 Summary

In this chapter we have evaluated the network performance of CONFIDANT fortified DSR. We first designed performance metrics and strategies which are the basis of the subsequent evaluations. Various performance metrics are defined: Good and evil throughput, evil drop rate, overhead, misbehavior identification rate and false negative/positive rate.

A large number of preliminary simulations have been conducted to adjust the CONFIDANT related parameters to achieve the best network performance. Apart from good/evil throughput, other metrics such as misbehavior identification rate and false negative rate are also used as criteria to select most appropriate values. Especially a method of step-by-step inference is employed to adjust some particular parameter.

In the main body of the chapter, CONFIDANT and standard DSR have been compared in aspects of good/evil throughput, evil drop rate and overhead. The results show that CONFIDANT has great improvement in evil throughput and evil drop rate. However, CONFIDANT does not increase the good throughput due to no enough good routes in the network. Furthermore, CONFIDANT increases routing overhead significantly.

We have also compared CONFIDANT with two other variations, Path re-ranking and Using trust. The results show that CONFIDANT is better at coping with misbehavior than Path re-ranking does. However, Using trust does not speed up the detection of misbehavior as expected. Due to the time limitation, we did not investigate the reason further.

8 Conclusion

This chapter concludes the thesis work in relation to the project of *Dynamic feed-back mechanisms in Trust-Based DSR*. In the conclusion we review the objective of thesis and summarize the main contribution of the thesis. During the thesis we also encountered some interesting problems that we do not have time to investigate. These problems are stated in the future work section.

8.1 Conclusion

In section 1.3 we have stated that the objective of the thesis is to investigate the dynamic feedback mechanisms as security solutions of Mobile Ad Hoc Network, to implement one of the mechanisms and evaluate its performance. To achieve this objective we designed several sub-tasks. This thesis has carried out the sub-tasks and completed the objective. The contribution of the thesis is as follows.

We have prepared preliminary information about the Mobile Ad Hoc Network routing protocols, Bayesian analysis and other miscellaneous technologies that one must know to understand the thesis work.

An investigation has been conducted on the state of the art technologies dealing with security issues in Mobile Ad Hoc Network. As a result, the general security issues/requirement of Mobile Ad Hoc Network has been summarized. The investigation also covers the security solutions of payment system, reputation system, trust-based system and a new intrusion diction system. We have compared the advantage and disadvantages of these systems.

Detailed analyses have been presented about DSR protocol, CONFIDANT protocol and network simulator. DSR has many additional features that will impact CONFIDANT and influence the network performance. We designed criteria and discussed each feature to decide whether it should be enabled or not. We described the details of CONFIDANT since it is the basis of implementing the protocol. We also discovered a problem of ns2 which has significant impact on the project and worked out a solution.

A large amount of simulations have been conducted to evaluate the performance of CONFIDANT fortified DSR. The simulation results show that CONFIDANT significantly decreases the evil throughput and evil drop rate by up to more than 50%. It proves that CONFIDANT can effectively mitigate misbehavior in the network. However CONFIDANT does not improve the good throughput due to the large increase of send buffer drop, which is caused because there are not enough good routes in the network.

It is worth to mention that during the performance analysis we developed step-by-step inference methods to address the problems. For example, it was used when we was estimating the value of misbehaved threshold and when we was investigating the reason why CONFIDANT had low good throughput.

We also evaluated the performance of two variations of CONFIDANT, Path re-ranking and Using trust. The simulation result shows that CONFIDANT is better at coping with misbehavior compared to Path re-ranking. However, Using trust does not speed up the detection of misbehavior as expected. We did not investigate the reason due to the time limitation.

8.2 Future Work

We have implemented and evaluated the performance of the CONFIDANT protocol based on DSR. During the thesis work we observed some results that are out of our expectation. We also encountered some problems that we feel worth to investigate but just have no time to complete. In this section we illustrate some of the topics that could be explored in future.

- Investigate the impact of DSR flow state on CONFIDANT. Flow state extension is a recent feature of DSR. CONFIDANT does not specify how to deal with it. However flow state extension could impact CONFIDANT since it allows the intermediate nodes to change the route based on their local knowledge of the network. That means the Passive Acknowledgement mechanism that CONFIDANT used to detect misbehavior may not work. In our thesis the feature is simply disabled. But it is worthwhile to investigate whether CONFIDANT could incorporate the feature since it is said the feature can improve the network throughput effectively.

- Investigate the performance of Using trust. Using trust to accept secondhand information is said to be able to speed up the misbehavior detection time. However, our simulation results show that using trust actually slows down the detection. More investigation about why the detection time of using trust is slow could be done.

- Investigate ns2 simulator regarding wireless links. The reason why we get very low good throughput for CONFIDANT is that there are not enough good routes available in the network. The ns2 could be investigated further about why there are few wireless links. If the wireless links could be increased then the good throughput of the CONFIDANT could be re-evaluated.

- Evaluate the network overhead of CONFIDANT considering the size of packet. In our thesis we only consider the number of the increased routing packet for overhead. However, the published information packets are usually much larger than other normal routing packets. Furthermore, the number of packets introduces different cost than the size of packets. Thus a better method could be designed to evaluate the network overhead.

A Bibliography

[1] David B. Johnson, David A. Maltz and Yih-Chun Hu. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR) for Mobile Ad Hoc Networks (DSR), draft version 10.

[2] Sean R Eddy. What is Bayesian Statistics.

[3] Bayesian analysis. http://mathworld.wolfram.com/BayesianAnalysis.html

[4] Bayesian logic.
http://whatis.techtarget.com/definition/0,,sid9_gci548993,00.html

[5] [ns] DSR performance is too bad in Ns2, why? *Ns-2 email archive*.
http://mailman.isi.edu/pipermail/ns-users/2004-March/040579.html

[6] The Network Simulator – ns2 homepage.
http://www.isi.edu/nsnam/ns/index.html

[7] Sonja Buchegger. Coping with Misbehavior in Mobile Ad-hoc Networks. February, 2004.

[8] Charles E. Perkins. Ad Hoc Network.

[9] Z.Yan and P. Cofta. Methodology to Bridge Different Domains of Trust. *Trust management first international conference, iTrust 2003*

[10] A. Josang, S.Hird, E.Faccer. Simulating the Effect of Reputation System. *Trust management first international conference, iTrust 2003*

[11] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, Jorjeta Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad Hoc Nework Routing Protocols.

[12] GloMoSim Global Mobile Information Systems Simulation Library homepage.
http://pcl.cs.ucla.edu/projects/glomosim/

[13] Raj Jain. The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.

[14] Lennart Conrad. M.Sc. Thesis Secure Routing in Mobile Ad Hoc Networks.

[15] Po-Wah Yau and Chris J.Mitchell. Security Vulnerabilities in Ad Hoc Networks.

[16] Frank Stajano and Ross Anderson . The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks.

[17] Levente Buttyan and Jean-Pierre Hubaux. Nuglets: a Virtual Currency to Stimulate Cooperation in Self-Organized Mobile Ad Hoc Networks.

[18] Levente Buttyan and Jean-Pierre Hubaux. Stimulating Cooperation in Self-Organizing Mobile Ad Hoc Networks.

[19] Sheng Zhong, Jiang Chen, Yang Richard Yang: Sprite: A Simple, Cheat-Proof, Credit-based System for Ad-Hoc Networks.

[20] Yongwei Wang, Venkata C. Giruka, Mukesh Singhal. A Fair Distributed Solution for Selfish Nodes Problem in Wireless Ad Hoc Networks

[21] Giorgos Zacharia, Alexandros Moukas and Pattie Maes. Collaborative Reputation Mechanisms in Electronic Marketplaces

[22] Sonja Buchegger, Jean-Yves Le Boudec. Nodes Bearing Grudges: Towards Routing Security, Fairness and Robustness in Mobile Ad Hoc Networks.

[23] Sonja Buchegger, Jean-Yves Le Boudec. The Effect of Rumor Spreading in Reputation Systems for Mobile Ad-hoc Networks.

[24] Pietro Michiardi and Refik Molva. CORE: A Collaborative Reputation Mechanism to enforce node cooperation in Mobile Ad hoc Networks.

[25] Sorav Bansal and Mary Baker. Observation-based Cooperation Enforcement in Ad hoc Networks.

[26] Jiangyi Hu Cooperation in Mobile Ad Hoc Networks

[27] Sergio Marti, T.J. Giuli, Kevin Lai, and Mary Baker. Mitigating Routing Misbehavior in Mobile Ad Hoc Networks.

[28] Yih-Chun Hu, Adrian Perrig, David B. Johnson: Ariadne: A Secure On-Demand Routing Protocol for Ad Hoc Networks.

[29] Kevin Fall, Kannan Varadhan. The ns Manual.

[30] OPNET Modeler. http://www.opnet.com/products/modeler/home.html

[31] David Cavin, Yoadv Sasson and Andre Schiper. On the Accuracy of MANET Simulators

[32] Slavisa Sarafijanovic and Jean/Yves le Boudec. An Artificial Immune System for Misbehavior Detection in Mobile Ad Hoc Networks with both Innate, Adaptive Subsystems and with Danger signal.

[33] P. Resnick and R. Zeckhauser. Trust among strangers in internet transactions: Empirical analysis of ebay’s reputation system.

B Acronyms

CBR	Constant Bit Rate
CONFIDANT	Cooperation Of Nodes: Fairness In Dynamic Ad-hoc Network
CORE	COllaborative REputation mechanism
DSR	Dynamic Source Routing
DoS	Denial of Service
LARS	Locally Aware Reputation System
MAC	Medium Access Control
MANET	Mobile Ad Hoc Network
NS	Network Simulator
OCEAN	Observation-based Cooperation Enforcement in Ad Hoc Networks
PACK	Passive Acknowledgement

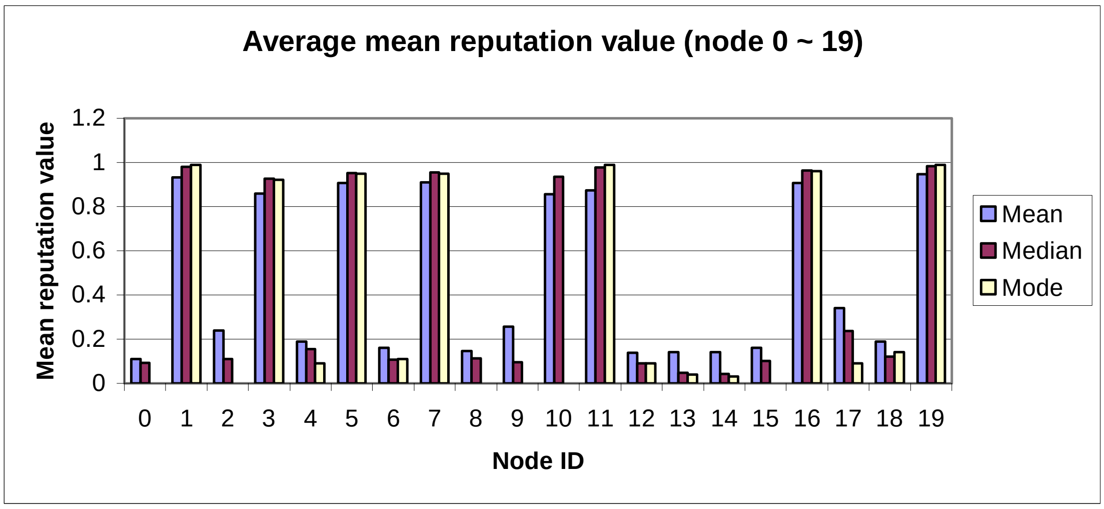
C List of Figures

D List of Tables

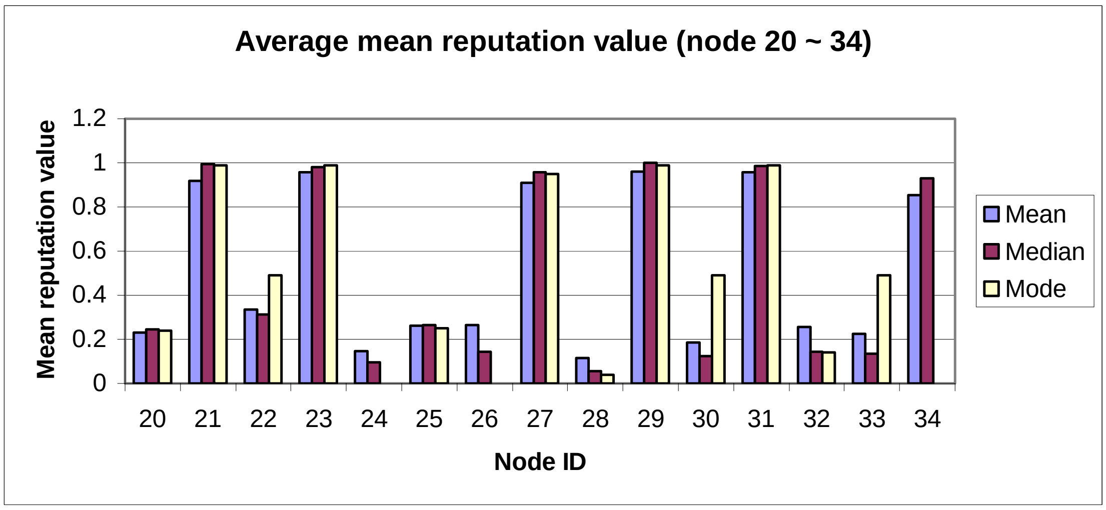
E List of Equations

F Average Mean Reputation Values

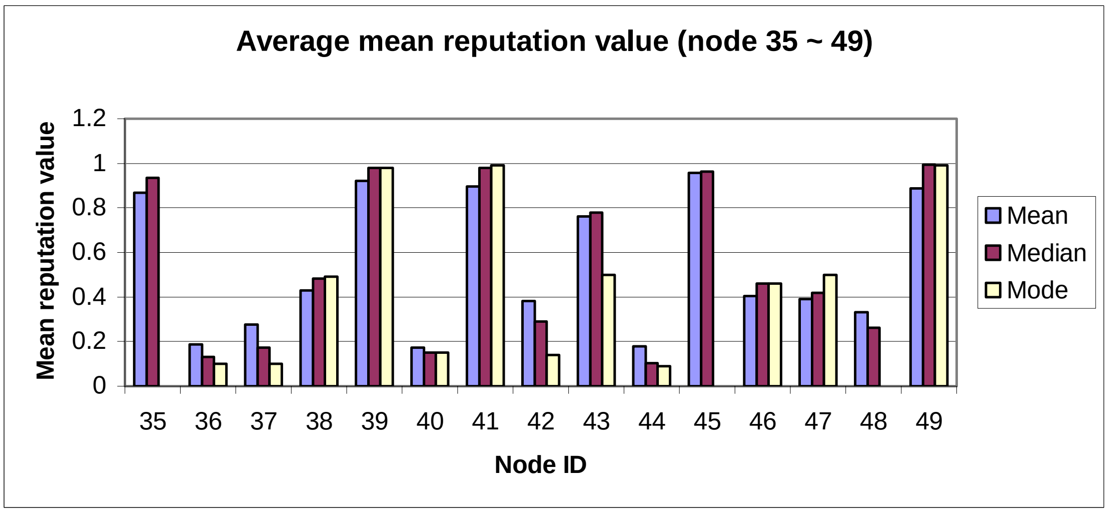
Following figures present the average mean reputation values about all the fifty nodes in the MANET. Three methods are used to calculate the average and the results show that the average mean reputation values are similar for these three methods. Thus each of the methods can be used to calculate the average.



Appendix Figure 1 Average mean reputation value of node 0 ~ 19



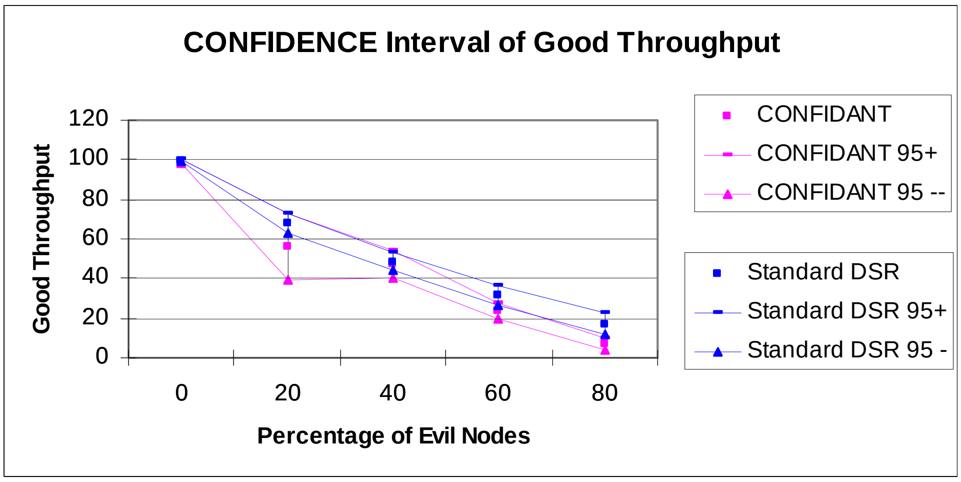
Appendix Figure 2 Average mean reputation value of node 20 ~ 34



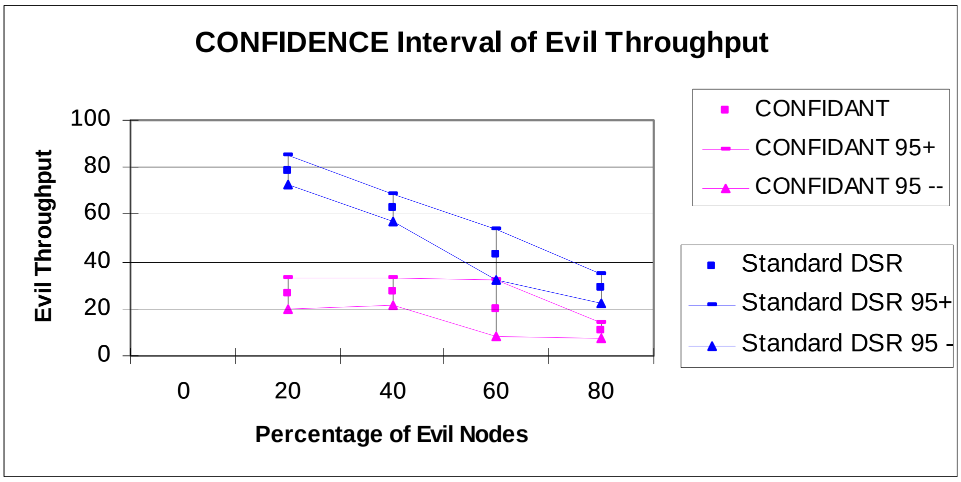
Appendix Figure 3 Average mean reputation value of node 35 ~ 49

G Confidence Interval Comparison

Following figures present the Confidence Interval of good throughput and evil throughput. The results show that the Confidence Intervals of good throughput for CONFIDANT and standard DSR have overlap. Thus CONFIDANT does not show any improvement over standard DSR (see Appendix Figure 4). However, CONFIDANT decreases evil throughput since its confidence interval is lower than that of standard DSR and does not overlap (see Appendix Figure 5).



Appendix Figure 4 CONFIDENCE interval of good throughput



Appendix Figure 5 CONFIDENCE interval of evil throughput

H Command of Creating Random Files

Command Used to Create Node-Movement Files:

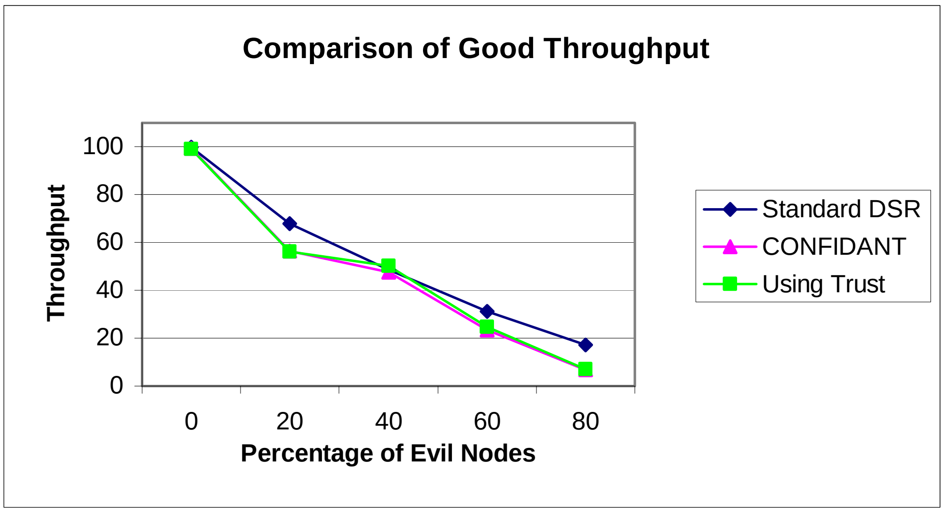
```
setdest -n 25 -p 60.0 -M 10 -t 500 -x 1000 -y 1000 > scen-
25-conf1
```

Command Used to Create Traffic Pattern Files:

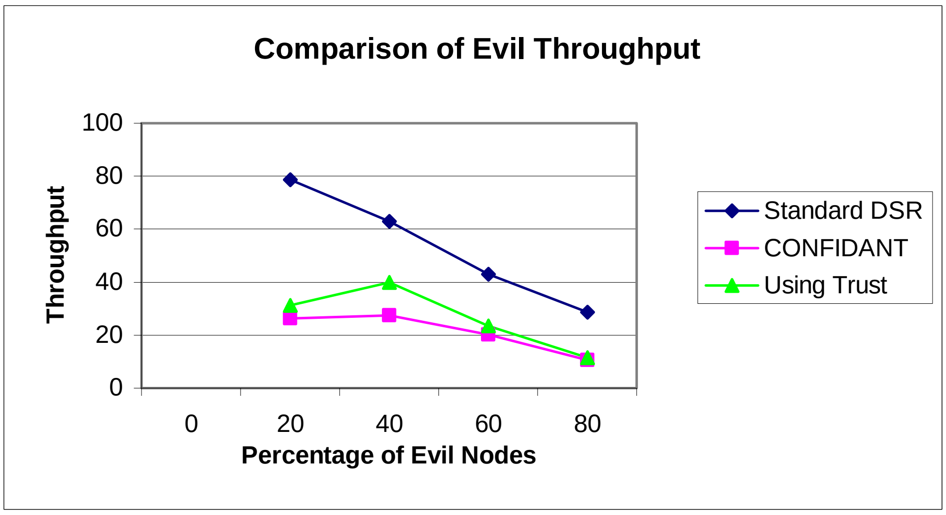
```
ns cbrgen.tcl -type cbr -nn 25 -seed 1.0 -mc 20 -rate 4.0 >
cbr-25-conf
```

I Simulation Results of Using Trust

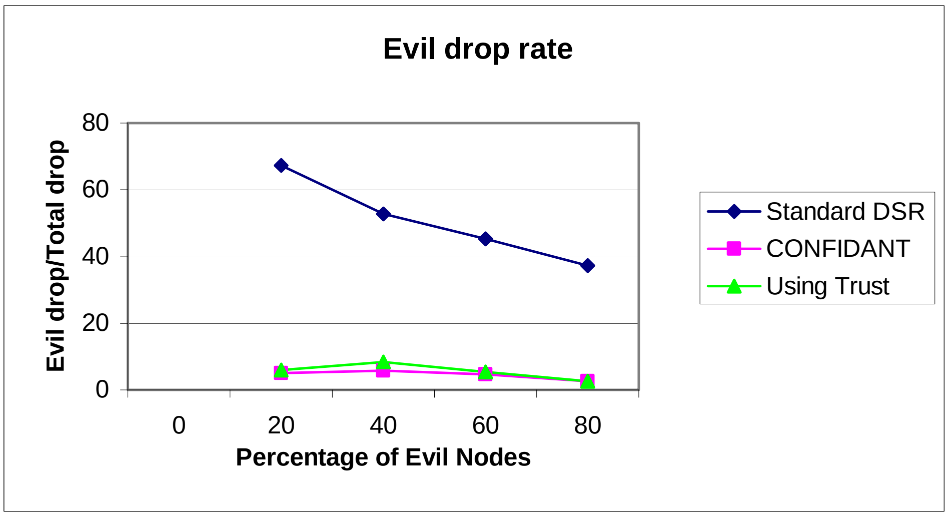
Following figures present the simulation results of using trust comparing with CONFIDANT and standard DSR. The results show that using trust does not improve the performance.



Appendix Figure 6 Comparison of good throughput



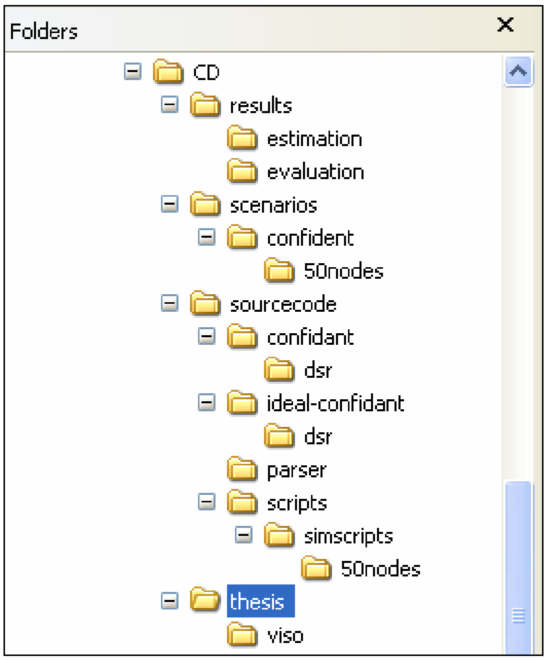
Appendix Figure 7 Comparison of evil throughput



Appendix Figure 8 Comparison of evil drop rate

J Content of CD

A CD containing all the material related to the project has been submitted with this thesis. Following screen shot presents the folder structure of the CD. The detailed installation guide can be seen in the README.txt file on the CD.



```
Folders
CD
    results
        estimation
        evaluation
    scenarios
        confident
            50nodes
    sourcecode
        confidant
            dsr
        ideal-confidant
            dsr
        parser
        scripts
            simscripts
                50nodes
    thesis
        viso
```

Appendix Figure 9 Folder structure of the CD

K Source Code

K.1 Hdr_Confidant.h

```
#ifndef _hdr_confdiant
#define _hdr_confdiant
#include "path.h"
#include <map>

#define NROFTOTALNODES 50
#define NROFEVILNODES 0
#define BEARGRUDGE 1
#define CONFIDANTVERBOSE 1
#define CONFIDANTDEBUG 0

/*** constants for monitor ***/
#define PACK_TIMEOUT 0.5
#define LOG_TIMEOUT 100

/*** constants for reputation system ***/
#define INACTIVITY_FADING 0.9
#define TRUST_FADING 0.9
#define DEVIATION_THRESHOLD 0.5
#define SECONDHAND_INFO_WEIGHT 0.2
#define MISBEHAVIOR_TOLERANCE 0.9
#define UNTRUST_TOLERANCE 0.95
#define INACTIVITY_TIMEOUT 2
#define PUBLISHING_TIMEOUT 3
#define USING_TRUST 1

enum Behavior {NOMATCH, GOOD, NOTFORWARDING};

class Rating
{
  public:
    Rating() { alpha = 1.0; beta = 1.0;}
    Rating(double a, double b) { alpha = a; beta = b; last_t =
Scheduler::instance().clock();}
    void updateRating(double a, double b, double fading, double weight);
    inline double getAlpha() { return alpha; }
    inline double getBeta() { return beta; }
    inline Time getTime() { return last_t; }
  private:
    //the rating of misbehavior
    double alpha;
    //the rating of good behavior
    double beta;
    //last updated time
    Time last_t;
};
```

```
class PackData {
  public:
    PackData();
    PackData(Packet* packet, Time t);
    Packet* packet;
    Time t;
};


typedef map<nsaddr_t, Rating*> RatingTable;
typedef map<int, PackData*> PackTable;

#endif
```

K.2 Monitor.h

```
#ifndef _monitor_h
#define _monitor_h

#include "dsragent.h"
#include "detector.h"
#include "hdr_confdiant.h"
#include "reputationsystem.h"
#include "routecache.h"
#include <map>

class DSRAgent;
class Detector;
class ReputationSystem;

class PackTableTimer : public TimerHandler {
public:
  PackTableTimer(Monitor *a) : TimerHandler() { a_ = a;}
  void expire(Event *e);
protected:
  Monitor *a_;
};

class Monitor {

public:
  Monitor();
  Monitor(DSRAgent* agent);
  void handleTap(const Packet* packet);
  //handle the tapped packet in promiscious mode. The function
  //will check whether the packet has been modified malicously
  //and report to Reputation System.
  void handlePublishInfo(nsaddr_t src, double identification, int count,
rating* ratings);
  //handle the first hand information packet sent by other nodes
  void handlePacketSent(Packet* packet);
  //the function insert the sent packet into the PACKTable and
  //associate the packet with a timout.
  void publishInfo(map<nsaddr_t, Rating*> ratings);
  //Reputation System call this function to send the first hand
  //information to neighboring nodes.
  void setNetID(nsaddr_t netid);
  nsaddr_t getNetID();
  void setReputationSystem(ReputationSystem* rep_system);
  bool isPACK(int uid);
  void Terminate();

private:
  void packTableCheck();
  map<nsaddr_t, double> published_ids;
  PackTable pack_t;
  DSRAgent* dsragent;
  Detector* detector;
  ReputationSystem* reputation_system;
  nsaddr_t net_id;
  PackTableTimer* pack_table_timer;
  friend class PackTableTimer;

};

#endif //_monitor_h
```

K.3 Monitor.cc

```
#include "monitor.h"
#include <scheduler.h>
#include <random.h>
#include <iostream>
#include <fstream>

ofstream monitorlog("monitorlog.txt");

void
PackTableTimer::expire(Event *e)
{
  a_->packTableCheck();
  resched(PACK_TIMEOUT + PACK_TIMEOUT * Random::uniform(1.0));
}

/***************************************************
 ****    Public functions of Monitor class     *****
 ***************************************************/
Monitor::Monitor(DSRAgent* agent)
{
  this->dsragent = agent;
  this->detector = new Detector(this);
  pack_table_timer = new PackTableTimer(this);
  pack_table_timer->sched(PACK_TIMEOUT
               + PACK_TIMEOUT * Random::uniform(1.0));
}

void Monitor::handleTap(const Packet* packet)
{
  Behavior behavior;
  hdr_sr *srh =  hdr_sr::access(packet);
  hdr_ip *iph =  hdr_ip::access(packet);
  hdr_cmn *cmh =  hdr_cmn::access(packet);

  ID cur_hop(srh->addrs()[srh->cur_addr()-1]);
  nsaddr_t cur_addr = cur_hop.getNSAddr_t();

  int uid = cmh->uid();
  map<int, PackData*>::iterator it;

  it = pack_t.find(uid);
  if (!(it == pack_t.end()))
  {
    //pass to Detector to check
    behavior = detector->detect(packet, pack_t[uid]);
    if (CONFIDANTDEBUG)
    {
      monitorlog <<"Node " << net_id << " handleTap uid " << uid << "
address " << cur_addr;
      monitorlog << "'s behavior " << behavior << endl;
    }
    reputation_system->handleFirstHandInfo(cur_addr, behavior);
    //is correct???
    pack_t.erase(it);

  }
  else
  {
    if (CONFIDANTDEBUG)
    monitorlog << "Node " << net_id << " handleTap cannot find uid " <<
uid << " in pack_t" << endl;
  }
  return;
}

void Monitor::handlePublishInfo(nsaddr_t from, double identification,
int count, rating* ratings)
{
  //handle duplicate identification here
  //...
  /*
  map<nsaddr_t, double>::iterator it;
  it = published_ids.find(from);
  if (it != published_ids.end() && (it->second == identification))
    {
    return;
    }*/
  if (CONFIDANTDEBUG)
  {
    monitorlog << "Node " << net_id << " receive publish info from " <<
from;
    monitorlog << " identification: " << identification << " count " <<
count;
  }

  for (int i = 0; i < count ; i ++, ratings ++)
  {
    if (CONFIDANTDEBUG)
    {
      monitorlog << " address " << ratings->addr << " alpha " <<
ratings->alpha;
      monitorlog << " beta " << ratings->beta << ", ";
    }
    if (ratings->addr != net_id)
      reputation_system->handleSecondHandInfo(from, ratings->addr,
ratings->alpha, ratings->beta);
  }
  if (CONFIDANTDEBUG)
    monitorlog << endl;
}

void Monitor::handlePacketSent(Packet* packet)
{
  Packet* copy_packet = packet->copy();

  hdr_cmn *cmh =  hdr_cmn::access(copy_packet);
  hdr_sr *srh = hdr_sr::access(copy_packet);
  hdr_ip *iph = hdr_ip::access(copy_packet);

  int uid = cmh->uid();

  map<int, PackData*>::iterator it;
  it = pack_t.find(uid);
  if (it != pack_t.end())
  {
    if (CONFIDANTDEBUG)
      monitorlog << "Node _" << net_id << "_ Big Error: The same packet
is sent again!" << endl;
    //pack_t.erase(it);
    return;
  }

  if (srh->route_request() || srh->publish_info() || (cmh->next_hop()
== iph->daddr()))
    return;
  if (CONFIDANTDEBUG)
    monitorlog << "Node _" << net_id << "_ add packet uid " << uid << "
into pack_t" << endl;
  pack_t[uid] = new PackData(copy_packet,
Scheduler::instance().clock());
}

void Monitor::setNetID(nsaddr_t netid)
{
  this->net_id = netid;
}

nsaddr_t Monitor::getNetID()
{
  return this->net_id;
}

bool Monitor::isPACK(int uid)
{
  map<int, PackData*>::iterator it;
  it = pack_t.find(uid);//if id doesn't exist it returns iterator to
the end
  bool result = !(it == pack_t.end());
  return result;
}

void Monitor::publishInfo(map<nsaddr_t, Rating*> ratings)
{
  dsragent->publishInfo(ratings);
}

void Monitor::setReputationSystem(ReputationSystem* rep_system)
{
  reputation_system = rep_system;
}

void Monitor::Terminate()
{
}

/****************************************************
 **** Private functions of Monitor class       *****
 ****************************************************/
void Monitor::packTableCheck()
{
  map<int, PackData*>::iterator it;
  for(it = pack_t.begin();it!=pack_t.end();it++)
  {
    if ((Scheduler::instance().clock() -(it->second)->t)>PACK_TIMEOUT)
    {
    hdr_cmn *cmh =  hdr_cmn::access(it->second->packet);
    //nsaddr_t pre_hop = cmh->prev_hop_;
    nsaddr_t next_hop = cmh->next_hop();
    if (CONFIDANTDEBUG || CONFIDANTVERBOSE)
    {
    monitorlog <<"Node " << net_id << " packTableCheck " << next_hop;
    monitorlog << " not forward packet uid " << it->first << endl;
    }
    reputation_system->handleFirstHandInfo(next_hop, NOTFORWARDING);
    pack_t.erase(it);
    }
  }
}
```

K.4 Reputationsystem.h

```
#ifndef _reputation_system
#define _reputation_system

#include "monitor.h"
#include "trustmanager.h"
#include "hdr_confdiant.h"
#include "dsragent.h"
#include <map>
```

```
class ReputationSystem;
class DSRAgent;

class InactivityTimer : public TimerHandler {
  public:
    InactivityTimer(ReputationSystem *a) : TimerHandler() { a_ = a;}
    void expire(Event *e);
  protected:
    ReputationSystem *a_;
};

class PublishingTimer : public TimerHandler {
  public:
    PublishingTimer(ReputationSystem *a) : TimerHandler() { a_ = a;}
    void expire(Event *e);
  protected:
    ReputationSystem *a_;
};

class ReputationSystem
{
  public:
    /****** functions ******/
    ReputationSystem(DSRAgent* agent);
    //constructor
    void handleFirstHandInfo(nsaddr_t& address, Behavior behavior);
    //Handle firsthand oberserved information
    void handleSecondHandInfo(nsaddr_t from, nsaddr_t address, double
alpha, double beta);
    //Handle secondhand information published by neighbor node
    void setPathManager(RouteCache* pathmanager);
    void setNetID(nsaddr_t address);
    void Terminate();

  private:

    /****** functions ******/
    void handlePublishingTimeout();
    void handleInactivityTimeout();

    Rating* initNewRating(double alpha, double beta, double fading);
    void updateReputation(nsaddr_t address, double alpha, double beta,
double fading, double weight);
    bool deviationTest(nsaddr_t address, double alpha1, double beta1,
double alpha2, double beta2);
    bool isMisbehavedNode(nsaddr_t address);
    double BayeMean(double alpha, double beta);

    /****** variables ******/
    DSRAgent* dsragent;
    RouteCache* path_manager;
    TrustManager trust_manager;
    map<nsaddr_t, Rating*> firsthandinfo_t;
    RatingTable reputation_t;
    InactivityTimer* inactivity_timer;
    PublishingTimer* publishing_timer;
    friend class InactivityTimer;
    friend class PublishingTimer;
    nsaddr_t net_id;
};
#endif
```

K.5 Reputationsystem.cc

```
#include "reputationsystem.h"
#include <scheduler.h>
#include <random.h>
#include <iostream>
#include <math.h>
#include <fstream>

ofstream reputationlog("reputationlog.txt");

void InactivityTimer::expire(Event * e)
{
  a_->handleInactivityTimeout();
  resched(INACTIVITY_TIMEOUT + INACTIVITY_TIMEOUT *
Random::uniform(1.0));
}

void PublishingTimer::expire(Event * e)
{
  a_->handlePublishingTimeout();
  resched(PUBLISHING_TIMEOUT + PUBLISHING_TIMEOUT *
Random::uniform(1.0));
}

/*****************************************************
 **** Public functions of ReputationSystem class ****
 *****************************************************/
ReputationSystem::ReputationSystem(DSRAgent* agent)
{
  this->dsragent = agent;
  this->inactivity_timer = new InactivityTimer(this);
  inactivity_timer->sched(INACTIVITY_TIMEOUT
             + INACTIVITY_TIMEOUT * Random::uniform(1.0));
  this->publishing_timer = new PublishingTimer(this);
  publishing_timer->sched(PUBLISHING_TIMEOUT
             + PUBLISHING_TIMEOUT * Random::uniform(1.0));
}

void ReputationSystem::handleFirstHandInfo(nsaddr_t& address, Behavior
behavior)
{
  short alpha = (behavior == GOOD) ? 0 : 1;
  map<nsaddr_t, Rating*>::iterator it;
  it = firsthandinfo_t.find(address);
  if (it == firsthandinfo_t.end())
  {
    if(CONFIDANTDEBUG)
    {
      reputationlog << "Node " << net_id << " add new address " <<
address;
      reputationlog << " to firsthandinfo table" << endl;
    }
    firsthandinfo_t[address] = initNewRating(alpha, 1-alpha, 1);
  }
  else
  {
    (it->second)->updateRating(alpha, 1-alpha, INACTIVITY_FADING, 1);
  }
  if(CONFIDANTDEBUG)
  {
    reputationlog << "Node " << net_id << " handle firsthand info from
";
    reputationlog << address << " alpha " << alpha << " beta " << 1-
alpha << endl;
  }
  updateReputation(address, alpha, 1-alpha, INACTIVITY_FADING, 1);
  return;
}

void ReputationSystem::handleSecondHandInfo(nsaddr_t from, nsaddr_t
address, double alpha1, double beta1)
{
  map<nsaddr_t, Rating*>::iterator it;
  it = reputation_t.find(address);
  if (it == reputation_t.end())
  {
  reputation_t[address] = new Rating(1.0, 1.0);
  }

  double alpha2 = reputation_t[address]->getAlpha();
  double beta2 = reputation_t[address]->getBeta();
  bool isPass = deviationTest(address, alpha1, beta1, alpha2, beta2);
  if ( ((USING_TRUST == 1) && isPass) ||
       ((USING_TRUST == 1) && trust_manager.isTrustworthy(from)))
  {
    if(CONFIDANTDEBUG)
    {
      reputationlog << "Node " << net_id << " handle secondhand info
from ";
      reputationlog << from << ":" << address << " [" << alpha1 << ","
<< beta1 << "] ";
      reputationlog << " [" << alpha2 << "," << beta2 << "] "  << endl;
    }
    updateReputation(address, alpha1, beta1, 1, SECONDHAND_INFO_WEIGHT);
  }
  double alpha = isPass ? 0 : 1;
  trust_manager.updateTrust(address, alpha, 1-alpha);
  return;
}

void ReputationSystem::setNetID(nsaddr_t address)
{
  net_id = address;
}
```

```
void ReputationSystem::setPathManager(RouteCache* pathmanager)
{
  this->path_manager = pathmanager;
}

void ReputationSystem::Terminate()
{
  map<nsaddr_t, Rating*>::iterator it;
  for (it = reputation_t.begin(); it != reputation_t.end(); it ++)
  {
    reputationlog << "_" <<net_id << "_ Reputation values [" << it-
>first << ", alpha:";
    reputationlog << it->second->getAlpha() << ", beta:" << it->second-
>getBeta() << "] ";
    reputationlog << "mean: " << BayeMean(it->second->getAlpha(), it-
>second->getBeta()) << endl;
  }
  path_manager->Terminate(net_id);
}

/******************************************************
 **** Private functions of ReputationSystem class ****
 ******************************************************/
Rating* ReputationSystem::initNewRating(double alpha, double beta,
double fading)
{
  Rating* rating = new Rating(1.0, 1.0);
  rating->updateRating(alpha, beta, fading, 1);
  return rating;
}

void ReputationSystem::updateReputation(nsaddr_t address, double alpha,
double beta, double fading, double weight)
{
  map<nsaddr_t, Rating*>::iterator it;
  it = reputation_t.find(address);
  if (it == reputation_t.end())
  {
  reputation_t[address] = initNewRating(alpha, beta, 1);
  }
  else
  {
    if (CONFIDANTDEBUG)
    {
      reputationlog << "Before updating: ";
      reputationlog << "_" << net_id <<"_ node "<<address <<": [";
      reputationlog << (it->second)->getAlpha() <<","<<(it->second)-
>getBeta()<<"]"<<endl;
    }
    (it->second)->updateRating(alpha, beta, fading, weight);
  }
  it = reputation_t.find(address);
  if (CONFIDANTDEBUG)
  {
    reputationlog << "After updating: ";
    reputationlog << "_" << net_id <<"_ node "<<address <<" reputation
updated: [";
    reputationlog << (it->second)->getAlpha() <<","<<(it->second)-
>getBeta()<<"]"<<endl;
  }
  if (isMisbehavedNode(address))
  {
    if (CONFIDANTDEBUG)
      reputationlog << "Node " << address << " is put into _" << net_id
<< "_ misbehaved list" << endl;
    path_manager->addMisbehavedNode(address);
  }
  else
  {
    path_manager->removeMisbehavedNode(address);
  }
  return;
}

void ReputationSystem::handleInactivityTimeout()
{
  map<nsaddr_t, Rating*>::iterator it;
  for (it = firsthandinfo_t.begin(); it != firsthandinfo_t.end(); it ++)
  {
    if ( (Scheduler::instance().clock() - (it->second)->getTime()) >
INACTIVITY_TIMEOUT)
    (it->second)->updateRating(0, 0, INACTIVITY_FADING, 1);
  }

  map<nsaddr_t, Rating*>::iterator it2;
  for (it2 = reputation_t.begin(); it2 != reputation_t.end(); it2 ++)
  {
    if ( (Scheduler::instance().clock() - (it2->second)->getTime()) >
INACTIVITY_TIMEOUT)
    (it2->second)->updateRating(0, 0, INACTIVITY_FADING, 1);
  }

  trust_manager.handleInactivityTimeout();
}

void ReputationSystem::handlePublishingTimeout()
{
  if (firsthandinfo_t.empty())
    return;

  if(CONFIDANTDEBUG)
  {
    map<nsaddr_t, Rating*>::iterator it;
    it = firsthandinfo_t.begin();
    reputationlog << "Node " << net_id << " at ";
    reputationlog << Scheduler::instance().clock() << "publish time out
";

    while (it != firsthandinfo_t.end())
    {
      double alpha = (it->second)->getAlpha();
      double beta = (it->second)->getBeta();
      reputationlog << "[" << it->first << ": " << alpha << "," << beta
<<"]";
      it ++;
    }
    reputationlog << endl;
  }

  dsragent->publishInfo(firsthandinfo_t);
}

bool ReputationSystem::deviationTest(nsaddr_t address, double alpha1,
double beta1, double alpha2, double beta2)
{
  if ((alpha1 + beta1) == 0)
    cout << "Big error! The rading of address " << address << "is zero";
  double d = (double) (alpha1/(alpha1+beta1)) -(double)
(alpha2/(alpha2+beta2));
  double dev = fabs(d);
  return (dev > DEVIATION_THRESHOLD) ? false:true;
}

double ReputationSystem::BayeMean(double alpha, double beta)
{
  return alpha/(alpha+beta);
}

bool ReputationSystem::isMisbehavedNode(nsaddr_t address)
{
  map<nsaddr_t, Rating*>::iterator it;
  it = reputation_t.find(address);
  if (it != reputation_t.end())
  {
    double alpha = (it->second)->getAlpha();
    double beta = (it->second)->getBeta();
    double dev = alpha/(alpha+beta);
    bool rlt = (dev >= MISBEHAVIOR_TOLERANCE) ? true:false;
    if (CONFIDANTDEBUG && rlt)
    {
           reputationlog << "Node " << address << " ["<<alpha << "," <<
beta << ",";
           reputationlog << dev << "] is misbehaved node" << endl;
    }
    return rlt;
  }
  else
  {
    if (CONFIDANTDEBUG)
         reputationlog << "Cannot find node " << address << endl;
    return false;
  }
}
```

K.6 Trustmanager.h

```
#ifndef _TRUST_MANAGER
```

```
#include "hdr_confdiant.h"
#include <map>

class TrustManager
{
  public:
    void updateTrust(nsaddr_t address, double alpha, double beta);
    bool isTrustworthy(nsaddr_t from);
    void handleInactivityTimeout();

  private:
    Rating* initNewRating(double alpha, double beta, double fading);
    map<nsaddr_t, Rating*> trust_t;
};
#endif
```

K.7 Trustmanager.cc

```
#include "trustmanager.h"

/*****************************************************
 ****    Public functions of TrustManager class  ****
 *****************************************************/
void TrustManager::updateTrust(nsaddr_t address, double alpha, double
beta)
{
  map<nsaddr_t, Rating*>::iterator it;
  it = trust_t.find(address);
  if (it == trust_t.end())
  {
    trust_t[address] = initNewRating(alpha, beta, 1);
  }
  else
  {
    (it->second)->updateRating(alpha, beta, TRUST_FADING, 1);
  }
}


bool TrustManager::isTrustworthy(nsaddr_t from)
{
  map<nsaddr_t, Rating*>::iterator it;
  it = trust_t.find(from);
  if (it != trust_t.end())
  {
    double alpha = (it->second)->getAlpha();
    double beta = (it->second)->getBeta();
    double dev = alpha/(alpha+beta);
    return (dev >= UNTRUST_TOLERANCE) ? false:true;
  }
  else
    return true;
}
```

```
void TrustManager::handleInactivityTimeout()
{
  map<nsaddr_t, Rating*>::iterator it;
  for (it = trust_t.begin(); it != trust_t.end(); it ++)
  {
    if ( (Scheduler::instance().clock() - (it->second)->getTime()) >
INACTIVITY_TIMEOUT)
    (it->second)->updateRating(0, 0, TRUST_FADING, 1);
  }
}

/*****************************************************
 ****   Private functions of TrustManager class   ****
 *****************************************************/
Rating* TrustManager::initNewRating(double alpha, double beta, double
fading)
{
  Rating* rating = new Rating(1.0, 1.0);
  rating->updateRating(alpha, beta, fading, 1);
  return rating;
}
```

K.8 Pathmanager.cc

```
extern "C" {
#include <stdio.h>
#include <stdarg.h>
};

#undef DEBUG

#include <scheduler.h>
#include <random.h>
#include <god.h>
#include "path.h"
#include "routecache.h"
#include "hdr_confdiant.h"
#ifdef DSR_CACHE_STATS
#include "cache_stats.h"
#endif
#include <iostream>
#include <set>
#include <fstream>

#define fout stdout

static const int verbose = 0;
static const int verbose_debug = 0;

ofstream pathmanagerlog("pathmanagerlog.txt");
ofstream misbehavenodeslog("misbehavenodeslog.txt");

/*=======================================================
  function selectors
```

```
---------------------------------------------------------*/
bool cache_ignore_hints = false;     // ignore all hints?
bool cache_use_overheard_routes = true;
// if we are A, and we over hear a rt Z Y (X) W V U, do we add route
// A X W V U to the cache?


/*=========================================================
  Class declaration
---------------------------------------------------------*/
class PathManager;

class Cache {
friend class PathManager;

public:
  Cache(char *name, int size, PathManager *rtcache);
  ~Cache();

  int pickVictim(int exclude = -1);
  // returns the index of a suitable victim in the cache
  // will spare the life of exclude
  bool searchRoute(const ID& dest, int& i, Path &path, int &index);
  // look for dest in cache, starting at index,
  //if found, rtn true with path s.t. cache[index] == path && path[i]
== dest
  Path* addRoute(Path &route, int &prefix_len);
  // rtns a pointer the path in the cache that we added
  void noticeDeadLink(const ID&from, const ID& to);
  // the link from->to isn't working anymore, purge routes containing
  // it from the cache

private:
  Path *cache;
  int size;
  int victim_ptr;       // next victim for eviction
  PathManager *routecache;
  char *name;
};

/////////////////////////////////////////////////////////////class
LogTimer : public TimerHandler {
      public:
        LogTimer(PathManager *a) : TimerHandler() { a_ = a;}
        void expire(Event *e);
      protected:
        PathManager *a_;
};

/////////////////////////////////////////////////////////////
class PathManager: public RouteCache
{
friend class Cache;
friend class MobiHandler;

public:
  PathManager(const ID& MAC_id, const ID& net_id, int psize = 30,int
ssize = 34 );
  PathManager();
  ~PathManager();
  void noticeDeadLink(const ID&from, const ID& to, Time t);
  // the link from->to isn't working anymore, purge routes containing
  // it from the cache
  void noticeRouteUsed(const Path& route, Time t,
                   const ID& who_from);
  // tell the cache about a route we saw being used
  void addRoute(const Path& route, Time t, const ID& who_from);
  // add this route to the cache (presumably we did a route request
  // to find this route and don't want to lose it)
  bool findRoute(ID dest, Path& route, int for_use = 0);
  // if there is a cached path from us to dest returns true and fills
in
  // the route accordingly. returns false otherwise
  // if for_use, then we assume that the node really wants to keep
  // the returned route so it will be promoted to primary storage if
not there
  // already
  int command(int argc, const char*const* argv);
  //CONFIDANT
  void addMisbehavedNode(nsaddr_t address);
  //add a misbehaved node to the misbehaved node list. This function is
called
  // by reputation manager.
  void removeMisbehavedNode(nsaddr_t address);
  //remove a node to the misbehaved node list. This function is called
  // by reputation manager.
  bool isNodeSafe(nsaddr_t address);
  // Is a node safe from the opinion of current node?
  bool isPathSafe(const Path& path);
  // Is a path safe from the opinion of current node?
  void checkDropReason(ID dest);
  // Check the reason why packets are dropped from send buffer. The
reason
  // could be no route in the route cache or only bad routes. This
function
  // is called by dsragent::dropSendBuff
  void Terminate(nsaddr_t id);
  // Do the clean up work when the simulation terminiate.

protected:
  Cache *primary_cache;   /* routes that we are using, or that we have
reason
                       to believe we really want to hold on to */
  Cache *secondary_cache; /* routes we've learned via a speculative
process
                       that might not pan out */

#ifdef DSR_CACHE_STATS
  void periodic_checkCache(void);
  void checkRoute(Path *p, int action, int prefix_len);
  void checkRoute(Path &p, int&, int&, double&, int&, int&, double &);
#endif
```

```
private:
  void handleLogTimeout();
  bool isNodeEvil(nsaddr_t id);  //is the node actually evil?
  bool isPathEvil(const Path& path); //does the path actually contain
evil node?
  multiset<nsaddr_t> misbehavednode_list;
  LogTimer* log_timer;
  friend class LogTimer;
};

RouteCache *
makeRouteCache()
{
  return new PathManager();
}


/*===========================================================
  OTcl definition
-----------------------------------------------------------*/
static class PathManagerClass : public TclClass {
public:
        PathManagerClass() : TclClass("PathManager") {}
        TclObject* create(int, const char*const*) {
                return (new PathManager);
        }
} class_PathManager;

/*===========================================================
 Constructors
-----------------------------------------------------------*/
PathManager::PathManager(): RouteCache()
{
  primary_cache = new Cache("primary", 60, this);
  secondary_cache = new Cache("secondary", 128, this);
  //secondary_cache = new Cache("secondary", 10000, this);
  assert(primary_cache != NULL && secondary_cache != NULL);
#ifdef DSR_CACHE_STATS
      stat.reset();
#endif
  this->log_timer = new LogTimer(this);
  log_timer->sched(LOG_TIMEOUT);
}

PathManager::~PathManager()
{
  delete primary_cache;
  delete secondary_cache;
}

int
PathManager::command(int argc, const char*const* argv)
{
  if(argc == 2 && strcasecmp(argv[1], "startdsr") == 0)
    {
      if (ID(1,::IP) == net_id)
      trace("Sconfig %.5f using PathManager",
Scheduler::instance().clock());
      // FALL-THROUGH
    }
  return RouteCache::command(argc, argv);
}

#ifdef DSR_CACHE_STATS

void
PathManager::periodic_checkCache()
{
  int c;
  int route_count = 0;
  int route_bad_count = 0;
  int subroute_count = 0;
  int subroute_bad_count = 0;
  int link_bad_count = 0;
  double link_bad_time = 0.0;
  int link_bad_tested = 0;
  int link_good_tested = 0;

  for(c = 0; c < primary_cache->size; c++)
    {
      int x = 0;

      if (primary_cache->cache[c].length() == 0) continue;

      checkRoute(primary_cache->cache[c],
                 x,
                 link_bad_count,
                 link_bad_time,
                 link_bad_tested,
                 link_good_tested,
             stat.link_good_time);

      route_count += 1;
      route_bad_count += x ? 1 : 0;

      subroute_count += primary_cache->cache[c].length() - 1;
      subroute_bad_count += x;
    }
  for(c = 0; c < secondary_cache->size; c++)
    {
      int x = 0;

      if (secondary_cache->cache[c].length() == 0) continue;

      checkRoute(secondary_cache->cache[c],
                 x,
                 link_bad_count,
                 link_bad_time,
                 link_bad_tested,
                 link_good_tested,
             stat.link_good_time);

      route_count += 1;
      route_bad_count += x ? 1 : 0;

      subroute_count += secondary_cache->cache[c].length() - 1;
      subroute_bad_count += x;
    }

  // lifetime of good link is (total time) / (total num links - num bad
links)
  stat.link_good_time = stat.link_good_time / (subroute_count -
link_bad_count);

  trace("SRC %.9f _%s_ cache-summary %d %d %d %d | %d %.9f %d %d | %d
%d %d %d %d | %d %d %d %d %d | %d %d %d %d %d %d %.9f",
      Scheduler::instance().clock(), net_id.dump(),
        route_count,
        route_bad_count,
        subroute_count,
        subroute_bad_count,

        link_bad_count,
        link_bad_count ? link_bad_time/link_bad_count : 0.0,
        link_bad_tested,
        link_good_tested,

        stat.route_add_count,
        stat.route_add_bad_count,
        stat.subroute_add_count,
        stat.subroute_add_bad_count,
        stat.link_add_tested,

        stat.route_notice_count,
        stat.route_notice_bad_count,
        stat.subroute_notice_count,
        stat.subroute_notice_bad_count,
        stat.link_notice_tested,

        stat.route_find_count,
      stat.route_find_for_me,
        stat.route_find_bad_count,
        stat.route_find_miss_count,
        stat.subroute_find_count,
        stat.subroute_find_bad_count,

      stat.link_good_time);
  stat.reset();
}

#endif /* DSR_CACHE_STATS */

/*===========================================================
  member functions
-----------------------------------------------------------*/

void
PathManager::addRoute(const Path& route, Time t, const ID& who_from)
// add this route to the cache (presumably we did a route request
// to find this route and don't want to lose it)
// who_from is the id of the routes provider
{
  Path rt;

  if(pre_addRoute(route, rt, t, who_from) == 0)
    return;

  // must call addRoute before checkRoute
  int prefix_len = 0;

#ifdef DSR_CACHE_STATS
  Path *p = primary_cache->addRoute(rt, prefix_len);
  checkRoute(p, ACTION_ADD_ROUTE, prefix_len);
#else
  (void) primary_cache->addRoute(rt, prefix_len);
#endif
}

void
PathManager::noticeDeadLink(const ID&from, const ID& to, Time)
  // the link from->to isn't working anymore, purge routes containing
  // it from the cache
{
  if(verbose_debug)
    trace("SRC %.9f _%s_ dead link %s->%s",
        Scheduler::instance().clock(), net_id.dump(),
        from.dump(), to.dump());

  primary_cache->noticeDeadLink(from, to);
  secondary_cache->noticeDeadLink(from, to);
  return;
}


void
PathManager::noticeRouteUsed(const Path& p, Time t, const ID& who_from)
// tell the cache about a route we saw being used
{
  Path stub;
  if(pre_noticeRouteUsed(p, stub, t, who_from) == 0)
    return;

  int prefix_len = 0;

#ifdef DSR_CACHE_STATS
  Path *p0 = secondary_cache->addRoute(stub, prefix_len);
  checkRoute(p0, ACTION_NOTICE_ROUTE, prefix_len);
#else
  (void) secondary_cache->addRoute(stub, prefix_len);
#endif
}

bool
PathManager::findRoute(ID dest, Path& route, int for_me)
// if there is a cached path from us to dest returns true and fills in
// the route accordingly. returns false otherwise
// if for_me, then we assume that the node really wants to keep
// the returned route so it will be promoted to primary storage if not
there
// already
{
  Path path;
  int min_index = -1;
  int min_length = MAX_SR_LEN + 1;
  int min_cache = 0;          // 2 == primary, 1 = secondary
  int index;
  int len;
  bool falseBad = false;      //whether false bad route exists

  assert(!(net_id == invalid_addr));

  index = 0;
  while (primary_cache->searchRoute(dest, len, path, index))
    {
      bool isSafe = isPathSafe(path);
      bool isEvil = isPathEvil(path);
      //CONFIDANT the node thinks the route is not safe but actually it
is good
      //and discarded by mistake
      if (!isSafe && !isEvil)
        falseBad = true;
      //CONFIDANT do not return routes containing evil nodes
      if (isSafe)
      {
        if (len < min_length)
       {
         min_cache = 2;
         min_length = len;
         route = path;
       }
      }
      index++;
    }

  index = 0;
  while (secondary_cache->searchRoute(dest, len, path, index))
    {
      bool isSafe = isPathSafe(path);
      bool isEvil = isPathEvil(path);
      //CONFIDANT the node thinks the route is not safe but actually it
is good
      //and discarded by mistake
      if (!isSafe && !isEvil)
        falseBad = true;
      //CONFIDANT do not return routes containing evil nodes
      if (isSafe)
      {
        if (len < min_length)
        {
          min_index = index;
          min_cache = 1;
          min_length = len;
          route = path;
        }
      }
      index++;
    }

  if (min_cache == 1 && for_me)
    { // promote the found route to the primary cache
      int prefix_len;

      primary_cache->addRoute(secondary_cache->cache[min_index],
prefix_len);

      // no need to run checkRoute over the Path* returned from
      // addRoute() because whatever was added was already in
      // the cache.

      //   prefix_len = 0
      //        - victim was selected in primary cache
      //        - data should be "silently" migrated from primary to
the
      //          secondary cache
      //   prefix_len > 0
      //        - there were two copies of the first prefix_len routes
      //          in the cache, but after the migration, there will be
      //          only one.
      //        - log the first prefix_len bytes of the secondary cache
      //          entry as "evicted"
      if(prefix_len > 0)
        {
          secondary_cache->cache[min_index].setLength(prefix_len);
#ifdef DSR_CACHE_STATS
          checkRoute_logall(&secondary_cache->cache[min_index],
                            ACTION_EVICT, 0);
#endif
        }
      secondary_cache->cache[min_index].setLength(0); // kill route
    }

  if (min_cache)
    {
      route.setLength(min_length + 1);
      if (verbose_debug)
      trace("SRC %.9f _%s_ $hit for %s in %s %s",
            Scheduler::instance().clock(), net_id.dump(),
            dest.dump(), min_cache == 1 ? "secondary" : "primary",
            route.dump());
#ifdef DSR_CACHE_STATS
      int bad = checkRoute_logall(&route, ACTION_FIND_ROUTE, 0);
      stat.route_find_count += 1;
      if (for_me) stat.route_find_for_me += 1;
      stat.route_find_bad_count += bad ? 1 : 0;
      stat.subroute_find_count += route.length() - 1;
      stat.subroute_find_bad_count += bad;
#endif
      //CONFIDANT statistic only
      if (isPathEvil(route))
            {
               //evil path selected
          pathmanagerlog << "NotOK: " << route[0].getNSAddr_t()<<" -->
"<<dest.getNSAddr_t()<< " Route: " << route.dump() <<endl;
         if (falseBad)
         {
           //good path discarded. Instead a evil path is selected.
           pathmanagerlog << "GD: " << route[0].getNSAddr_t()<<" -->
"<<dest.getNSAddr_t()<< " Route: " << route.dump() <<endl;
         }
            }
      else
      {
        //good path selected
        pathmanagerlog << "OK: " << route[0].getNSAddr_t()<<" -->
"<<dest.getNSAddr_t()<< " Route: " << route.dump() <<endl;
            }

      return true;
    }
  else
    {
      if (verbose_debug)
        trace("SRC %.9f _%s_ find-route [%d] %s->%s miss %d %.9f",
              Scheduler::instance().clock(), net_id.dump(),
              0, net_id.dump(), dest.dump(), 0, 0.0);
#ifdef DSR_CACHE_STATS
      stat.route_find_count += 1;
      if (for_me) stat.route_find_for_me += 1;
      stat.route_find_miss_count += 1;
#endif
      return false;
    }
}

/*===========================================================
  class Cache routines
-----------------------------------------------------------*/

Cache::Cache(char *name, int size, PathManager *rtcache)
{
  this->name = name;
  this->size = size;
  cache = new Path[size];
  routecache = rtcache;
  victim_ptr = 0;
}

Cache::~Cache()
{
  delete[] cache;
}

bool
Cache::searchRoute(const ID& dest, int& i, Path &path, int &index)
  // look for dest in cache, starting at index,
  //if found, return true with path s.t. cache[index] == path && path[i]
== dest
{
  for (; index < size; index++)
    for (int n = 0 ; n < cache[index].length(); n++)
      if (cache[index][n] == dest)
      {
        i = n;
        path = cache[index];
        return true;
      }
  return false;
}

Path*
Cache::addRoute(Path & path, int &common_prefix_len)
{
  int index, m, n;
  int victim;

  // see if this route is already in the cache
  for (index = 0 ; index < size ; index++)
    { // for all paths in the cache
      for (n = 0 ; n < cache[index].length() ; n ++)
      { // for all nodes in the path
        if (n >= path.length()) break;
        if (cache[index][n] != path[n]) break;
      }
      if (n == cache[index].length())
      { // new rt completely contains cache[index] (or cache[index] is
empty)
          common_prefix_len = n;
          for ( ; n < path.length() ; n++)
            cache[index].appendToPath(path[n]);
        if (verbose_debug)
          routecache->trace("SRC %.9f _%s_ %s suffix-rule (len %d/%d)
%s",
            Scheduler::instance().clock(), routecache->net_id.dump(),
              name, n, path.length(), path.dump());
        goto done;
      }
      else if (n == path.length())
      { // new route already contained in the cache
          common_prefix_len = n;
        if (verbose_debug)
          routecache->trace("SRC %.9f _%s_ %s prefix-rule (len %d/%d)
%s",
            Scheduler::instance().clock(), routecache->net_id.dump(),
            name, n, cache[index].length(), cache[index].dump());
        goto done;
      }
      else
      { // keep looking at the rest of the cache
      }
    }

  // there are some new goodies in the new route
  victim = pickVictim();
  if(verbose_debug) {
    routecache->trace("SRC %.9f _%s_ %s evicting %s",
                  Scheduler::instance().clock(), routecache-
>net_id.dump(),
                  name, cache[victim].dump());
    routecache->trace("SRC %.9f _%s_ while adding %s",
                  Scheduler::instance().clock(), routecache-
>net_id.dump(),
                  path.dump());
  }
  cache[victim].reset();
  CopyIntoPath(cache[victim], path, 0, path.length() - 1);
  common_prefix_len = 0;
  index = victim; // remember which cache line we stuck the path into

done:

#ifdef DEBUG
  {
    Path &p = path;
    int c;
    char buf[1000];
    char *ptr = buf;
    ptr += sprintf(buf,"Sdebug %.9f _%s_ adding ",
               Scheduler::instance().clock(), routecache-
>net_id.dump());
    for (c = 0 ; c < p.length(); c++)
      ptr += sprintf(ptr,"%s [%d %.9f] ",p[c].dump(), p[c].link_type,
p[c].t);
    routecache->trace(buf);
  }
#endif //DEBUG

  // freshen all the timestamps on the links in the cache
  for (m = 0 ; m < size ; m++)
    { // for all paths in the cache

#ifdef DEBUG
  {
    if (cache[m].length() == 0) continue;

    Path &p = cache[m];
    int c;
    char buf[1000];
    char *ptr = buf;
    ptr += sprintf(buf,"Sdebug %.9f _%s_ checking ",
               Scheduler::instance().clock(), routecache-
>net_id.dump());
    for (c = 0 ; c < p.length(); c++)
      ptr += sprintf(ptr,"%s [%d %.9f] ",p[c].dump(), p[c].link_type,
p[c].t);
    routecache->trace(buf);
  }
#endif //DEBUG

      for (n = 0 ; n < cache[m].length() - 1 ; n ++)
      { // for all nodes in the path
        if (n >= path.length() - 1) break;
        if (cache[m][n] != path[n]) break;
        if (cache[m][n+1] == path[n+1])
          { // freshen the timestamps and type of the link

#ifdef DEBUG
routecache->trace("Sdebug %.9f _%s_ freshening %s->%s to %d %.9f",
              Scheduler::instance().clock(), routecache->net_id.dump(),
              path[n].dump(), path[n+1].dump(), path[n].link_type,
              path[n].t);
#endif //DEBUG

            cache[m][n].t = path[n].t;
            cache[m][n].link_type = path[n].link_type;
            /* NOTE: we don't check to see if we're turning a TESTED
             into an UNTESTED link.  Last change made rules -dam
5/19/98 */
          }
      }
    }
  return &cache[index];
}


void
Cache::noticeDeadLink(const ID&from, const ID& to)
  // the link from->to isn't working anymore, purge routes containing
  // it from the cache
{
  for (int p = 0 ; p < size ; p++)
    { // for all paths in the cache
      for (int n = 0 ; n < (cache[p].length()-1) ; n ++)
      { // for all nodes in the path
        if (cache[p][n] == from && cache[p][n+1] == to)
          {
            if(verbose_debug)
            routecache->trace("SRC %.9f _%s_ %s truncating %s %s",
                              Scheduler::instance().clock(),
                              routecache->net_id.dump(),
                              name, cache[p].dump(),
                              cache[p].owner().dump());
#ifdef DSR_CACHE_STATS
              routecache->checkRoute(&cache[p], ACTION_CHECK_CACHE, 0);
              routecache->checkRoute_logall(&cache[p],  ACTION_DEAD_LINK,
n);
#endif
            if (n == 0)
            cache[p].reset();        // kill the whole path
            else {
            cache[p].setLength(n+1); // truncate the path here
                cache[p][n].log_stat = LS_UNLOGGED;
              }

            if(verbose_debug)
            routecache->trace("SRC %.9f _%s_ to %s %s",
                  Scheduler::instance().clock(), routecache-
>net_id.dump(),
                  cache[p].dump(), cache[p].owner().dump());
            break;
          } // end if this is a dead link
      } // end for all nodes
    } // end for all paths
  return;
}

int
Cache::pickVictim(int exclude)
// returns the index of a suitable victim in the cache
// never return exclude as the victim, but rather spare their life
{
  for(int c = 0; c < size ; c++)
    //CONFIDANT select a bad route as victim
    if ((cache[c].length() == 0) || !routecache->isPathSafe(cache[c]))
      return c;

  int victim = victim_ptr;
  while (victim == exclude)
    {
      victim_ptr = (victim_ptr+1 == size) ? 0 : victim_ptr+1;
      victim = victim_ptr;
    }
  victim_ptr = (victim_ptr+1 == size) ? 0 : victim_ptr+1;

#ifdef DSR_CACHE_STATS
  routecache->checkRoute(&cache[victim], ACTION_CHECK_CACHE, 0);
  int bad = routecache->checkRoute_logall(&cache[victim], ACTION_EVICT,
0);
  routecache->trace("SRC %.9f _%s_ evicting %d %d %s",
                    Scheduler::instance().clock(), routecache-
>net_id.dump(),
                    cache[victim].length() - 1, bad, name);
#endif
  return victim;
}

#ifdef DSR_CACHE_STATS

/*
 * Called only for the once-per-second cache check.
 */
void
PathManager::checkRoute(Path & p,
                      int & subroute_bad_count,
                      int & link_bad_count,
                      double & link_bad_time,
                      int & link_bad_tested,
                      int & link_good_tested,
                  double & link_good_time)
{
  int c;
  int flag = 0;

  if(p.length() == 0)
    return;
  assert(p.length() >= 2);
  for (c = 0; c < p.length() - 1; c++)
    {
      assert(LS_UNLOGGED == p[c].log_stat || LS_LOGGED ==
p[c].log_stat );
      if (God::instance()->hops(p[c].getNSAddr_t(),
p[c+1].getNSAddr_t()) != 1)
      { // the link's dead
          if(p[c].log_stat == LS_UNLOGGED)
          {
            trace("SRC %.9f _%s_ check-cache [%d %d] %s->%s dead %d
%.9f",
                Scheduler::instance().clock(), net_id.dump(),
                p.length(), c, p[c].dump(), p[c+1].dump(),
                    p[c].link_type, p[c].t);
            p[c].log_stat = LS_LOGGED;
          }
          if(flag == 0)
            {
              subroute_bad_count += p.length() - c - 1;
              flag = 1;
            }
          link_bad_count += 1;
          link_bad_time += Scheduler::instance().clock() - p[c].t;
          link_bad_tested += (p[c].link_type == LT_TESTED) ? 1 : 0;
      }
      else
        {

        link_good_time += Scheduler::instance().clock() - p[c].t;

          if(p[c].log_stat == LS_LOGGED)
            {
              trace("SRC %.9f _%s_ resurrected-link [%d %d] %s->%s dead
%d %.9f",
                    Scheduler::instance().clock(), net_id.dump(),
                    p.length(), c, p[c].dump(), p[c+1].dump(),
                    p[c].link_type, p[c].t);
              p[c].log_stat = LS_UNLOGGED;
            }
          link_good_tested += (p[c].link_type == LT_TESTED) ? 1 : 0;
        }
    }
}

void
PathManager::checkRoute(Path *p, int action, int prefix_len)
{
  int c;
  int subroute_bad_count = 0;
  int tested = 0;

  if(p->length() == 0)
    return;
  assert(p->length() >= 2);

  assert(action == ACTION_ADD_ROUTE ||
         action == ACTION_CHECK_CACHE ||
         action == ACTION_NOTICE_ROUTE);

  for (c = 0; c < p->length() - 1; c++)
    {
      if (God::instance()->hops((*p)[c].getNSAddr_t(),
                                (*p)[c+1].getNSAddr_t()) != 1)
      { // the link's dead
          if((*p)[c].log_stat == LS_UNLOGGED)
            {
              trace("SRC %.9f _%s_ %s [%d %d] %s->%s dead %d %.9f",
                    Scheduler::instance().clock(), net_id.dump(),
                    action_name[action], p->length(), c,
                    (*p)[c].dump(), (*p)[c+1].dump(),
                    (*p)[c].link_type, (*p)[c].t);

              (*p)[c].log_stat = LS_LOGGED;
            }

          if(subroute_bad_count == 0)
            subroute_bad_count = p->length() - c - 1;
      }
      else
        {
          if((*p)[c].log_stat == LS_LOGGED)
            {
              trace("SRC %.9f _%s_ resurrected-link [%d %d] %s->%s dead
%d %.9f",
                    Scheduler::instance().clock(), net_id.dump(),
                    p->length(), c, (*p)[c].dump(), (*p)[c+1].dump(),
                    (*p)[c].link_type, (*p)[c].t);
              (*p)[c].log_stat = LS_UNLOGGED;
            }
        }
      tested += (*p)[c].link_type == LT_TESTED ? 1 : 0;
    }

  /*
   * Add Route or Notice Route actually did something
   */
  if(prefix_len < p->length())
    {
      switch(action)
        {
        case ACTION_ADD_ROUTE:
          stat.route_add_count += 1;
          stat.route_add_bad_count += subroute_bad_count ? 1 : 0;
          stat.subroute_add_count += p->length() - prefix_len - 1;
          stat.subroute_add_bad_count += subroute_bad_count;
          stat.link_add_tested += tested;
          break;

        case ACTION_NOTICE_ROUTE:
          stat.route_notice_count += 1;
          stat.route_notice_bad_count += subroute_bad_count ? 1 : 0;
          stat.subroute_notice_count += p->length() - prefix_len - 1;
          stat.subroute_notice_bad_count += subroute_bad_count;
          stat.link_notice_tested += tested;
          break;
        }
    }
}
#endif /* DSR_CACHE_STATS */

void PathManager::addMisbehavedNode(nsaddr_t address)
{
  if (misbehavednode_list.find(address) == misbehavednode_list.end())
  {
     //cout << "*****node " << address << " is put in misbehaved list";
     misbehavednode_list.insert(misbehavednode_list.begin(), address);
   }
}

void PathManager::removeMisbehavedNode(nsaddr_t address)
{
  multiset<nsaddr_t>::iterator mit;
  mit = misbehavednode_list.find(address);
  if (mit != misbehavednode_list.end())
  {
      //cout << "*****node " << address << " is removed from misbehaved
list";
      misbehavednode_list.erase(mit);
  }
}

bool PathManager::isNodeSafe(nsaddr_t address)
{
  return (misbehavednode_list.find(address) ==
misbehavednode_list.end());
}

bool PathManager::isPathSafe(const Path& path)
{
  Path mypath = path.copy();
  mypath.resetIterator();

  while (mypath.index() < mypath.length())
      {
        ID id = mypath.next();
        if (misbehavednode_list.find(id.getNSAddr_t()) !=
misbehavednode_list.end())
            return false;
      }
  return true;
}

void PathManager::checkDropReason(ID dest)
{
  Path path;
  int index;
  int len;
  bool goodRoute = false;
  bool badRoute = false;
  bool falseBad = false;
  assert(!(net_id == invalid_addr));

  index = 0;
  while (primary_cache->searchRoute(dest, len, path, index))
    {
      if (isPathSafe(path))
            {
              goodRoute = true;
        break;
            }
      else
            {
        badRoute = true;
        if (!isPathEvil(path))
        {
            falseBad = true;
        }
            }
      index++;
    }
  if (!goodRoute)
      {
        index = 0;
        while (secondary_cache->searchRoute(dest, len, path, index))
          {
           if (isPathSafe(path))
                  {
                    goodRoute = true;
              break;
                  }
            else
                  {
              badRoute = true;
              if (!isPathEvil(path))
              {
                  falseBad = true;
              }
                  }
            index++;
          }
      }

  if (goodRoute)
  {
    pathmanagerlog << "DROP_GOOD_ROUTE: " << net_id.getNSAddr_t() << "
-- " << dest.getNSAddr_t() << endl;
  }
  else if (! (goodRoute || badRoute))
  {
    pathmanagerlog << "DROP_NO_ROUTE: " << net_id.getNSAddr_t() << " --
" << dest.getNSAddr_t() << endl;
  }
  else if (!goodRoute && badRoute)
  {
    pathmanagerlog << "DROP_BAD_ROUTE " << net_id.getNSAddr_t() << " --
" << dest.getNSAddr_t() << endl;
```

```
    if (falseBad)
      pathmanagerlog << "DROP_FALSE_BAD_ROUTE " << net_id.getNSAddr_t()
<< " -- " << dest.getNSAddr_t() << endl;
  }
  else
  {
    pathmanagerlog << "DROP_UNKNOWN " << net_id.getNSAddr_t() << " -- "
<< dest.getNSAddr_t();
    pathmanagerlog << " [" << goodRoute << "," << badRoute << "," <<
falseBad << "]" << endl;
  }
}

void PathManager::Terminate(nsaddr_t id)
{
  multiset<nsaddr_t>::iterator it;
  misbehavenodeslog << "_" << id << "_ at "
<<Scheduler::instance().clock()<< " Misbehaved nodes are: ";
  for (it = misbehavednode_list.begin(); it !=
misbehavednode_list.end(); it ++)
  {
    misbehavenodeslog << *it << ", ";
  }
  misbehavenodeslog << endl;
}

bool PathManager::isNodeEvil(nsaddr_t id)
{
      //evil nodes id are in this array
      //for now they are hardcoded but they should be read from a file
      nsaddr_t* evilnodes;

      if (NROFTOTALNODES == 50)
      {
        nsaddr_t nodes[] = {1,3,7,11,16,21,27,34,43,45,
                        5,10,19,23,29,31,35,39,41,49,
                        2,8,13,17,25,26,32,37,42,47,
                        0,9,12,14,20,24,30,33,40,48};//max of 40
        evilnodes = nodes;
      }
      else if (NROFTOTALNODES == 40)
      {
        nsaddr_t nodes[] =
{1,3,7,10,13,16,17,19,23,26,29,30,33,36,37,39};//max of 16
        evilnodes = nodes;
      }
      else if (NROFTOTALNODES == 30)
      {
        nsaddr_t nodes[] = {1,3,7,10,13,16,17,19,23,26,29,30};//max of
12
        evilnodes = nodes;
      }

      for(int i = 0; i<NROFEVILNODES;evilnodes++, i++)
      {
        if(id == *evilnodes)
            {
                  //cerr<< "Node :"<< id << " is evil in Trust DSR"
<<endl;
                  return true;
            }
      }
  return false;
}

bool PathManager::isPathEvil(const Path& path)
{
  Path mypath = path.copy();
  mypath.resetIterator();
  while (mypath.index() < mypath.length())
      {
        ID id = mypath.next();
        if (isNodeEvil(id.getNSAddr_t()))
            return true;
      }
  return false;
}
void PathManager::handleLogTimeout()
{
  Terminate(net_id.getNSAddr_t());
}
void LogTimer::expire(Event * e)
{
  a_->handleLogTimeout();
  resched(LOG_TIMEOUT);
}

//#endif
```

K.9 DSRParser.java

```
import java.io.*;
import java.util.*;
import java.util.regex.PatternSyntaxException;

//Used to parse Trace files
public class DSRParser {
  public static int NROFNODES = 0;
  public static int NROFEVILNODES = 0;

  public DSRParser(){
  }

  /*
  **  THe methods reads from the file given as argument
  */
  public void parseTrace(String filename) {
    try
    {
      FileReader in = new FileReader(filename);
      BufferedReader reader = new BufferedReader(in);
      String line;
      int i = 0;           //nr of lines
      int s = 0;           //number of sent data packets
      int r =0;            //number of received data packets
      int D = 0;           //drop old format
      int SRReq = 0;       //number of route request sent
      int SRRly = 0;       //number of route reply sent
      int ERRly = 0;       //evil drop of route reply
      int EDATA = 0;       //number of evil drop of data packet
      int ERE = 0;         //evil drop of route error
      int ED = 0;          //other evil drop
      int RRR = 0;         //receive route reply
      int DEVIL = 0;       //drop data packets from evil nodes
      int DXMIT = 0;     //drop data packets due to link error
      int Ssb = 0;         //send buffer drop
      int ES = 0;          //send data packet from evil nodes
      int GS = 0;        //send data packet from good nodes
      int ER = 0;        //receive data packet from evil nodes
      int GR = 0;        //receive data packet from good nodes
      int SRR = 0;         //send route reply
      int S = 0;//Send DSR

      int D_RTR_NRTE = 0;   //drop due to no route
      int D_RTR_TOUT = 0;   //drop due to packet time out
      int D_RTR_SAL = 0;    //drop due to salvage
      int D_RTR_CBK = 0;    //drop due to mac callback
      int D_RTR_TTL = 0;
      int D_RTR_ROUTE_LOOP = 0;
      int D_RTR_END = 0;    //drop due to termination
      int D_IFQ_IFQ = 0;    //drop due to ifq full
      int D_IFQ_ARP = 0;    //drop due to arp
      int D_IFQ_END = 0;    //drop due to ifq termination
      int DDATA = 0;        //drop of data packet

      int nodedrop[] = new int[NROFNODES];

      for (int j = 0; j < nodedrop.length; j ++)
        nodedrop[j] = 0;

      line = reader.readLine();
      while(line != null) {
        line = line.trim();
          if (!line.equals("")) {
            StringTokenizer st = new StringTokenizer(line);
            String value = st.nextToken(" ");

            if(value.equals("D")) {
              //old format D
              //several drops can occur when the simulation ends
              //and these are not counted
              D++;
              String[] strs = line.split(" ");
              if ((strs.length > 5) && line.indexOf("cbr") > 0) {
                if (strs[3].equals("RTR") && strs[4].equals("NRTE")) {
                  D_RTR_NRTE ++;
                  DDATA ++;
                } else if (strs[3].equals("RTR") &&
strs[4].equals("TOUT")) {
                  D_RTR_TOUT ++;
                  DDATA ++;
                } else if (strs[3].equals("RTR") &&
strs[5].equals("SAL")) {
                  D_RTR_SAL ++;
                  DDATA ++;
                } else if (strs[3].equals("RTR") &&
strs[5].equals("CBK")) {
                    D_RTR_CBK ++;
                    DDATA ++;
                } else if (strs[3].equals("RTR") &&
strs[5].equals("TTL")) {
                  D_RTR_TTL ++;
                  DDATA ++;
                } else if (strs[3].equals("RTR") &&
strs[5].equals("END")) {
                    D_RTR_END ++;
                    DDATA ++;
                } else if (strs[3].equals("RTR") &&
strs[5].equals("LOOP")) {
                  D_RTR_ROUTE_LOOP ++;
                    DDATA ++;
                } else if (strs[3].equals("IFQ") &&
strs[5].equals("IFQ")) {
                  D_IFQ_IFQ ++;
                  DDATA ++;
                } else if (strs[3].equals("IFQ") &&
strs[5].equals("ARP")) {
                  D_IFQ_ARP ++;
                  DDATA ++;
                } else if (strs[3].equals("IFQ") &&
strs[5].equals("END")) {
                  D_IFQ_END ++;
                  DDATA ++;
                }
              }
            } else if(value.equals("EDATA")) {
              EDATA++;
              String[] data = line.split(" ");
              String str = data[2].substring(1);
              String str2 = str.substring(0, str.length()-1);
              int nodenr = Integer.parseInt(str2);
              nodedrop[nodenr]++;
            } else if(value.equals("ED")) {
              ED++;
            } else if(value.equals("SRReq")) {
              SRReq++;
            } else if(value.equals("SRRly")) {
              SRRly++;
            } else if(value.equals("ERRly")) {
              ERRly++;
            } else if(value.equals("RRR")) {
              RRR++;
            } else if(value.equals("ERE")) {
              ERE++;
            } else if(value.equals("DEVIL")) {
              DEVIL++;
            } else if(value.equals("DXMIT")) {
```

```
            DXMIT++;
          } else if(value.equals("Ssb")) {
            Ssb++;
          } else if(value.equals("ESend")) {
            ES++;
          } else if(value.equals("GSend")) {
            GS++;
          } else if(value.equals("ERecv")) {
            ER++;
          } else if(value.equals("GRecv")) {
            GR++;
          } else if(value.equals("SRR")) {
            SRR++;
          } else if(value.equals("s")) {
            s++;
          } else if(value.equals("S")) {
            S++;
          } else if(value.equals("r")) {
            r++;
          } else {
          }
        }
      i++;//we read 1 more line
      line = reader.readLine();
    }

    System.out.println("Read: " + i + " lines");
    System.out.println("D (old format): " + D + " ");
    System.out.println("s: " + s + " ");
    System.out.println("r: " + r + " ");
    System.out.println("SRReq: " + SRReq + " ");
    System.out.println("SRRly: " + SRRly + " ");
    System.out.println("ERRly: " + ERRly + " ");
    System.out.println("EDATA: "+ EDATA + " ");
    System.out.println("ERE: " + ERE + " ");
    System.out.println("ED: " + ED + " ");
    System.out.println("RRR: " + RRR + " ");
    System.out.println("DEVIL: " + DEVIL + " ");
    System.out.println("DXMIT: " + DXMIT + " ");
    System.out.println("Ssb: " + Ssb + " ");
    System.out.println("SRR: " + SRR);
    System.out.println("ESend: " + ES + " ");
    System.out.println("GSend: " + GS + " ");
    System.out.println("ERecv: " + ER + " ");
    System.out.println("GRecv: " + GR + " ");
    System.out.println("DDATA: " + DDATA + " ");
    System.out.println("D_RTR_NRTE: " + D_RTR_NRTE + " ");
    System.out.println("D_RTR_TOUT: " + D_RTR_TOUT + " ");
    System.out.println("D_RTR_SAL: " + D_RTR_SAL + " ");
    System.out.println("D_RTR_CBK: " + D_RTR_CBK + " ");
    System.out.println("D_RTR_TTL: " + D_RTR_TTL + " ");
    System.out.println("D_RTR_ROUTE_LOOP: " + D_RTR_ROUTE_LOOP + " ");
    System.out.println("D_RTR_END: " + D_RTR_END + " ");
    System.out.println("D_IFQ_IFQ: " + D_IFQ_IFQ + " ");
    System.out.println("D_IFQ_ARP: " + D_IFQ_ARP + " ");
    System.out.println("D_IFQ_END: " + D_IFQ_END + " ");
```

```
      double percent = ((s-r)*100)/s;
      System.out.println("Percentage dropped of send: "+percent+" ");
      if (GS != 0)
        System.out.println("Good node throughput: "+ GR*100/GS);
      if (ES != 0)
        System.out.println("Evil node throughput: "+ ER*100/ES);

      System.out.println("");
      System.out.println("Nodes dropped packets: ");
      for (int k = 0; k < nodedrop.length; k ++)  {
        System.out.println("node " + k + ": " + nodedrop[k]);
      }

      reader.close();
    }
    catch(IOException ie)
    {
      System.out.println("Usage: DSRParser trace filename");
      //System.out.println(ie.getMessage());
    }
  }

    public void parseRouteStats(String fileName) {
      try {
        FileReader in = new FileReader(fileName);
        BufferedReader reader = new BufferedReader(in);
        String line = "";
        int count = 0;
        //the resson of ssb drop
        int drop_total = 0;
        int drop_good_route = 0;
        int drop_no_route = 0;
        int drop_bad_route = 0;
        int drop_false_bad_route = 0;
        int drop_unknown = 0;
        //statistics of route selection
        int good_route_selected = 0;
        int good_route_discarded = 0;
        int false_good_selected = 0;

        line = reader.readLine();
        while(line != null) {
          line = line.trim();
          if (!line.equals("")) {
            count ++;
            StringTokenizer st = new StringTokenizer(line);
            String value = st.nextToken(": ");
            if (value.equals("DROP_GOOD_ROUTE")) {
              drop_good_route ++;
              drop_total ++;
            } else if (value.equals("DROP_NO_ROUTE")) {
              drop_no_route ++;
              drop_total ++;
            } else if (value.equals("DROP_BAD_ROUTE")) {
              drop_bad_route ++;
              drop_total ++;
            } else if (value.equals("DROP_FALSE_BAD_ROUTE")) {
              drop_false_bad_route ++;
            } else if (value.equals("DROP_UNKNOWN")) {
              drop_unknown ++;
              drop_total ++;
            } else if (value.equals("OK")) {
              good_route_selected ++;
            } else if (value.equals("NotOK")) {
              false_good_selected ++;
            } else if (value.equals("GD")) {
              good_route_discarded ++;
            }
          }
          line = reader.readLine();
        }
        System.out.println("DROP_GOOD_ROUTE: " + drop_good_route);
        System.out.println("DROP_NO_ROUTE: " + drop_no_route);
        System.out.println("DROP_BAD_ROUTE: " + drop_bad_route);
        System.out.println("DROP_FALSE_BAD_ROUTE: " +
drop_false_bad_route);
        System.out.println("DROP_UNKNOWN: " + drop_unknown);
        System.out.println("Total drop: " + drop_total);
        System.out.println();
        System.out.println("Good route selected: " +
good_route_selected);
        System.out.println("Bad route selected: " +
false_good_selected);
        System.out.println("Good route discarded: " +
good_route_discarded);
        int total_selection = good_route_selected + false_good_selected;
        System.out.println("Total selection: " + total_selection);
        System.out.println();
        System.out.println(count + " lines are parsed.");

        reader.close();
      }  catch(IOException ie) {
        System.out.println("Usage: DSRParser routestats filename");
        //System.out.println(ie.getMessage());
      }
    }

    public void parseReputationMean(String fileName) {
      try {
        FileReader in = new FileReader(fileName);
        BufferedReader reader = new BufferedReader(in);
        String line = "";
        int count = 0;
        int nomatch = 0;
        double[][] reputations = new double[NROFNODES][NROFNODES];
        double[] averageMean = new double[NROFNODES];
        double[] median = new double[NROFNODES];
        double[] mode = new double[NROFNODES];

        //init reputation mean values
        for (int i = 0; i < NROFNODES; i ++) {
          for (int j = 0; j < NROFNODES; j ++) {
            reputations[i][j] = -1.0;
          }
          averageMean[i] = -1.0;
        }

        String fnumber = "([0-9])+.?([0-9])*[e,E]?([+,-])?([0-9])*";
        String pattern = "_([0-9])+_ Reputation values \\p{Punct}([0-
9])+\\p{Punct} alpha:" + fnumber +
                "\\p{Punct} beta:" + fnumber + "\\p{Punct} mean: " +
fnumber;

        line = reader.readLine();
        while(line != null) {
          count ++;
          line = line.trim();
          if (!line.matches(pattern)) {
            //System.out.println("Strings not match! " + "line " +
count + ":" + line);
            nomatch ++;
            line = reader.readLine();
            continue;
          }

          int hostNode;
          int custNode;
          double mean;
          int leftSquare;
          int meanStart;

          hostNode  =  Integer.parseInt(line.substring(1,line.indexOf("_",
1)));
          leftSquare = line.indexOf("[");
          custNode = Integer.parseInt(line.substring(leftSquare+1,
line.indexOf(",", leftSquare)));
          meanStart = line.indexOf("mean: ") + 6;
          mean = Double.parseDouble(line.substring(meanStart));

          if (hostNode > NROFNODES || custNode > NROFNODES)
            throw new Exception("Node number out of range!");

          reputations[hostNode][custNode] = mean;
          line = reader.readLine();
        }
        System.out.println("Average mean reputation of each node:");
        for (int j = 0; j < NROFNODES; j ++) {
          int n = 0;
          double total = 0;
          for (int i = 0; i < NROFNODES; i ++) {
            if (reputations[i][j] < 0) {
              continue;
            }
            n ++;
            total += reputations[i][j];
          }
          averageMean[j] = total/n;
          //System.out.println("Node " + j + ": " + averageMean[j]);
          System.out.println(averageMean[j]);
        }
        System.out.println("\nMedian mean reputation of each node:");
        for (int j = 0; j < NROFNODES; j ++) {
          double[] sorted = sort(reputations, j);
          int n = 0;
          for (int i = 0; i < NROFNODES; i ++) {
            if (reputations[i][j] < 0) {
              continue;
            }
            n ++;
          }

          if ((n != 0) && (n%2 == 0)) {
            int n2 = n/2;
            median[j] = (sorted[n2-1]+sorted[n2])/2;
          } else {
            median[j] = sorted[n/2];
          }
          System.out.println(median[j]);
        }

        System.out.println("\nMode mean reputation of each node:");
        for (int j = 0; j < NROFNODES; j ++) {
          Object[] modelist = mode(reputations, j);
          for (int i = 0; i < modelist.length; i ++) {
            if (i != 0) {
              System.out.print(",");
            }
            System.out.print(((Double)modelist[i]).doubleValue());
          }
          System.out.println();
        }

        System.out.println();
        System.out.println("Detailed mean reputation of a node saved in
each node:");
        for (int j = 0; j < NROFNODES; j ++) {
          System.out.println("<<<Node " + j + ">>>");
          for (int i = 0; i < NROFNODES; i ++) {
            if (reputations[i][j] < 0) {
              System.out.println("X");
            } else {
              System.out.println(reputations[i][j]);
            }
          }
          System.out.println();
        }
      }  catch(IOException ie) {
        System.out.println("Usage: DSRParser reputation filename");
        //System.out.println(ie.getMessage());
      } catch (PatternSyntaxException pe){
        pe.printStackTrace();
      } catch (Exception e){
        e.printStackTrace();
      }
    }

    public void parseMisbihaviorIdentify(String fileName) {
      try {
        FileReader in = new FileReader(fileName);
        BufferedReader reader = new BufferedReader(in);
        String line = "";
        int count = 0;
        int nomatch = 0;
        double[] correctPercent = new double[9];
        double[] wrongPercent = new double[9];
        int correctCount = 0;
        int wrongCount = 0;
        int curTime = 1;

        for (int i = 0; i < 9; i ++) {
          correctPercent[i] = 0.0;
          wrongPercent[i] = 0.0;
        }

        String pattern = "_([0-9])+_ at ([0-9])+ Misbehaved nodes are:
(([0-9])+, )*";
        line = reader.readLine();
        while(line != null) {
          count ++;
          //line = line.trim();
          if (!line.matches(pattern)) {
            System.out.println("Strings not match! " + "line " + count
+ ":" + line);
            nomatch ++;
            line = reader.readLine();
            continue;
          }

          int timeStart;
          int timeEnd;
          int time;
          int evilNodesStart;
          String[] evilNodes;
          timeStart = line.indexOf("at ") + 3;
          timeEnd = line.indexOf(" ", timeStart);
          time = Integer.parseInt(line.substring(timeStart, timeEnd));
          if (time > curTime*100) {
            //Record percentage of correct/wrong identify at a specific
time
            //Then reset the counters.
            correctPercent[curTime-1] =
((correctCount*100)/NROFNODES)/NROFEVILNODES;
            wrongPercent[curTime-1] =
((wrongCount*100)/NROFNODES)/NROFEVILNODES;
            correctCount = 0;
            wrongCount = 0;
            curTime = time/100;
          }

          if (time <= curTime*100) {
            evilNodesStart = line.indexOf("Misbehaved nodes are: ") +
22;
            String str = line.substring(evilNodesStart);
            evilNodes = str.split(",");
            for (int i = 0; i < evilNodes.length; i ++) {
              evilNodes[i] = evilNodes[i].trim();
              if (!evilNodes[i].equals("")) {
                int nr = Integer.parseInt(evilNodes[i]);
                if (isEvil(nr, NROFEVILNODES)) {
                  correctCount ++;
                } else if (nr < NROFNODES) {
                  wrongCount ++;
                }
              }
            }
          }
          line = reader.readLine();
        }

        correctPercent[curTime-1] =
((correctCount*100)/NROFNODES)/NROFEVILNODES;
        wrongPercent[curTime-1] =
((wrongCount*100)/NROFNODES)/NROFEVILNODES;
        correctCount = 0;
        wrongCount = 0;

        for (int i = 0; i < 9; i ++) {
          int time = (i + 1) * 100;
          System.out.println("Identify rate at " + time + " is:");
          System.out.println("Correct: " + correctPercent[i]);
          System.out.println("Wrong: " + wrongPercent[i]);
        }
        System.out.println();
        System.out.println("Total line number is " + count);
        System.out.println("Not matched line number is " + nomatch);

      }  catch(IOException ie) {
          System.out.println("Usage: DSRParser reputation filename");
          //System.out.println(ie.getMessage());
      }
    }

    /* a sort algorithm. After sort the largest value is at the
beginning
     * of the sorted array and smallest value is at the end.
     */
    private double[] sort(double[][] originValues, int column) {
      double[] sorted = new double[NROFNODES];
      for (int i = 0; i < NROFNODES; i ++) {
        sorted[i] = originValues[i][column];
      }
      for (int i = 0; i < NROFNODES -1; i ++) {
        for (int j = 0; j < NROFNODES -1 - i; j ++) {
          double temp;
          if (sorted[j] < sorted[j+1]) {
            temp = sorted[j];
            sorted[j] = sorted[j+1];
            sorted[j+1] = temp;
          }
        }
      }
      return sorted;
    }

    private Object[] mode(double[][] originValues, int column) {
      ArrayList modes = new ArrayList();
      int[][] newValues = new int[NROFNODES][2];
      for (int i = 0; i < NROFNODES; i ++) {
        newValues[i][0] = (int)(originValues[i][column]*100);
        newValues[i][1] = -1; //mark
      }

      for (int i = 0; i < NROFNODES; i ++) {
        //skip those have the mean value of -1.0 or have been
calculated
        if ((newValues[i][0] == -100) || (newValues[i][1] != -1)) {
          continue;
        }
        int value = newValues[i][0];
        int count = 0;
        for (int j = 0; j < NROFNODES; j ++) {
          if (value == newValues[j][0]) {
            count ++;
            newValues[j][1] = 0; // marked as haven being calculated
          }
        }
        newValues[i][1] = count;
      }
      int max = 0;
      for (int i = 0; i < NROFNODES; i ++) {
        if (newValues[i][1] > max) {
          max = newValues[i][1];
        }
      }
      for (int i = 0; i < NROFNODES; i ++) {
        if (newValues[i][1] == max) {
          modes.add(new Double((double)newValues[i][0]/100.0));
        }
      }
      return modes.toArray();
    }

    private boolean isEvil(int id, int NROFEVILNODES) {
      //int[] evilnodes = {1,2,7,10,16,19,23,13,24,5};//max of 10

      int[] evilnodes = {1,3,7,11,16,21,27,34,43,45,
        5,10,19,23,29,31,35,39,41,49,
        2,8,13,17,25,26,32,37,42,47,
        0,9,12,14,20,24,30,33,40,48};//max of 40

      for(int i = 0; i<NROFEVILNODES;i++) {
        if(id == evilnodes[i]) {
          return true;
        }
      }
      return false;
    }
    private int getEvilPos(int id, int NROFEVILNODES) {
      int[] evilnodes = {1,3,7,11,16,21,27,34,43,45,
        5,10,19,23,29,31,35,39,41,49,
        2,8,13,17,25,26,32,37,42,47,
        0,9,12,14,20,24,30,33,40,48};//max of 40

      for(int i = 0; i<NROFEVILNODES;i++) {
        if(id == evilnodes[i]) {
          return i;
        }
      }
      return -1;
    }

  public static void main(String args[]) throws Exception
  {
    System.out.println("DSRParser!");
    if (args.length < 4) {
        throw new Exception("Wrong parameters! Please use command " +
                        "java DSRParser [parse type] [file name] [totl
nodes] [evil nodes]\n" +
                        "parse type: trace, routestats, reputation,
misbehave\n");

    }

    NROFNODES = Integer.parseInt(args[2]);
    NROFEVILNODES = Integer.parseInt(args[3]);

    String filename = args[1];
    DSRParser dsr = new DSRParser();
    if (args[0].equals("trace"))
      dsr.parseTrace(filename);
    else if (args[0].equals("routestats"))
      dsr.parseRouteStats(filename);
    else if (args[0].equals("reputation"))
      dsr.parseReputationMean(filename);
    else if (args[0].equals("misbehave")) {
      dsr.parseMisbihaviorIdentify(filename);
    }
    else {
        System.out.println("Wrong parameters! Please use command "+
                "java DSRParser [parse type] [file name] [nr of evil
nodes]");
      System.out.println("parse type: trace, routestats, reputation,
misbehave");
    }
  }
}
```

K.10 Runtestscript.tcl

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

#===========================================================
# Define options
```

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

# ------------------- simulation with 25 nodes ------------
set val(nn)            50 ;# number of mobilenodes
set val(x)       1000                 ;# X dimension of the topography
set val(y)             1000                 ;# Y dimension of the
topography
set val(stop)          900.0       ;# simulation time
set val(path)          /home/s031001/projects/NetSim/ns-allinone-
2.28/ns-2.28

#The cbr pattern is defined in this file and assiociated with cb
#30 connections
set val(cp)           "$val(path)/tcl/ex/confdiant/50nodes/cbr-50-r2";

#The scenario (nodes movement and connections) is defined in this file
and assiociated with sc
set val(sc)            "$val(path)/tcl/ex/confdiant/50nodes/scen-50-
m1-1";

#===========================================================
Agent/Null set sport_        0
Agent/Null set dport_        0

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

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

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

#Phy/WirelessPhy set bandwidth_ 11e6
#Mac/802_11 set basicRate_ 0
#Mac/802_11 set dataRate_ 0
#Mac/802_11 set bandwidth_ 11e6 ;
#Mac/802_11 set PLCPDataRate_ 11e6;


#===========================================================
# Main Program
#===========================================================
#Create a simulator object
set ns_           [new Simulator]

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

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

#Open the nam file
set namtrace [open confout.nam w]
$ns_ namtrace-all-wireless $namtrace $val(x) $val(y)

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

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

# Create God
create-god $val(nn)

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


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

```
#to the channel.


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

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

#Define node movement model


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

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

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

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

   $ns_ initial_node_pos $node_($i) 50
}

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

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

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

puts "Starting Simulation..."

$ns_ run
```

K.11 Run.sh

```
#!/bin/sh

export CLASSPATH=.

for i in "1" "2" "3" "4" "5"
do
```

```
    echo "running simulation $i..."
    cd ns-allinone-2.28/ns-2.28
    ./ns simscripts/$1nodes/rundsrscript$i.tcl
    mv conf-out-tdsr.tr ../../parser
    mv monitorlog.txt ../../parser/r2/evil$2/scen$i
    mv reputationlog.txt ../../parser/r2/evil$2/scen$i
    mv pathmanagerlog.txt ../../parser/r2/evil$2/scen$i
    mv misbehavenodeslog.txt ../../parser/r2/evil$2/scen$i
    cd ../../parser
    java DSRParser trace conf-out-tdsr.tr $1 $2 >
r2/evil$2/scen$i/scen-r2-$i.txt
    java DSRParser routestats r2/evil$2/scen$i/pathmanagerlog.txt $1 $2
>> r2/evil$2/scen$i/scen-r2-$i.txt
    cd ..
done
```

K.12 Runsim.sh

```
#!/bin/sh

export CLASSPATH=.

#./run1.sh

for i in "0" "10" "20" "30" "40"
do
    echo "Simulating evil nodes no. $i ..."
    cp params/50nodes/hdr_confdiant.hevil$i ns-allinone-2.28/ns-
2.28/dsr/hdr_confdiant.h
    cd ns-allinone-2.28/ns-2.28
    rm dsr/*.o
    make
    cd ../..
    ./run.sh 50 $i
done
```