Sphinx: Locating Malicious Nodes in Corporate Distributed Hash Tables

Yoram Versluis

TU Delft

Delft University of Technology

Sphinx: Locating Malicious Nodes in Corporate Distributed Hash Tables

Master's Thesis in Computer Science

Parallel and Distributed Systems Group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Yoram Versluis

24th April 2009

Author
Yoram Versluis

Title
Sphinx: Locating Malicious Nodes in Corporate Distributed Hash Tables

MSc presentation
29th April 2009

Graduation Committee

Prof.dr.ir. H.J. Sips	Delft University of Technology
Dr.ir. D.H.J. Epema	Delft University of Technology
Dr. P. Cimiano	Delft University of Technology
B.Y. Zhao, PhD	University of California Santa Barbara

Abstract

Nowadays structured overlay networks are used in controlled environments like company wide area networks. Even though these structured overlay networks are usually closed, there is still the risk that one or more nodes get compromised by attackers. When these networks provide business critical services, serious harm can be done by a compromised node. Because these networks are company controlled, properties like available bandwidth and latency times are much better known than in a general wide area network like the Internet. This knowledge can be used to create a system where malicious nodes in the network can be detected and located in a relatively short period of time and with little overhead. This thesis describes the design and implementation of the Sphinx protocol that provides this functionality. We show that it is possible to detect and locate malicious nodes in a distributed hash table in a short period of time, with little overhead, and with high sensitivity. We are able to detect and locate the nodes that cause latency deviations, dropping of messages, mis-routing of messages, and changing of the message payload.

Preface

This document describes my MSc project on detecting and locating malicious nodes in corporate distributed hash tables. This research was started at the CURRENT Lab of the University of California Santa Barbara under the supervision of Ben Y. Zhao, and completed at the Parallel and Distributed Systems Group of Delft University of Technology under the supervision of Dick Epema.
First of all, I would like to thank my supervisors Ben Zhao and Dick Epema for their guidance during this MSc project. I would further like to thank prof. dr. ir. H.J. Sips for chairing the examination committee, and dr. P. Cimiano for participating in the examination committee. Last, but definitely not least, I would like to thank Deborah for reviewing my thesis and for her patience and moral support during the whole period of this MSc project. Thank you.

Yoram Versluis

Delft, The Netherlands
24th April 2009

Contents

Chapter 1

Introduction

Peer-to-Peer (P2P) networks are decentralised networks where no node has a complete overview of the network, i.e., nodes can find eachother and communicate with each other with no central server involved. P2P networks became popular with the sharing of files. One of the first real, i.e., completely decentralised P2P protocols deployed and widely used was Gnutella. Gnutella however, was a non-structured overlay network. Non-structured overlay networks cannot provide guarantees whether stored objects can be found in the network. Structured overlays on the other hand allow a node to find another node or object in a deterministic way in relatively few steps. Distributed hash tables (DHTs) are a class of structured overlays where (key,value) pairs are stored in a distributed way. DHTs provide a way to reliably store and retrieve data in a non-centralised way. With no single point of failure the data are also highly available. For this reason, companies start deploying DHTs in corporate environments to provide reliable business critical services. For instance, Amazon created the Dynamo system, a DHT [2] for highly available key-value storage, to be used for their web shop. By using a DHT, Amazon is able to deliver the optimal user experience: quick response times and high availability.

These DHTs usually run in a company owned and controlled wide area networks (WAN). Even though these DHTs run in closed networks, history teaches us that even closed networks may get intruded by non-authorised people. Although the odds are low that the network gets intruded, the consequences may be severe when it happens. When an intruder is able to replace nodes with specially crafted nodes to disrupt the traffic in the DHT, serious harm can be done. Critical messages can be dropped or delayed, disrupting the reliability and/or speed of the DHT. Hence, we need a solution to cope with these attacks on DHTs running in corporate environments. Fortunately, the fact that the DHT runs in a company controlled WAN, gives us a number of advantages: network properties, like latency and round trip times, are much better known than the network properties of general WANs like the Internet. With this information, it is easier to determine whether latency is "natural" or deliberately caused.

This thesis describes the design and implementation of the Sphinx algorithm. The

Sphinx algorithm successfully detects malicious behaviour, and locates the nodes that causes this behaviour, within a DHT, in a scalable way and with a reasonable amount of overhead. The experiments conducted show that it is possible to detect and locate malicious behaviour with a high sensitivity. The Sphinx algorithm is able to detect and locate malicious nodes causing message droppage, forward delays, misrouting and tampering with the message's payload.

This thesis is structured as follows. In Chapter 2 we will give an overview of the functioning of DHTs. We will discuss key-based-routing and distributed hash tables, and the Chimera protocol upon which the Sphinx algorithm is implemented. We will also discuss the environments in which our targeted DHTs reside. We will end this chapter with the problem statement. In Chapter 3 we present and explain how the Sphinx algorithm detects and locates malicious behaviour. We will describe the use of signed acknowledgements as "proofs of innocence". We will present a reputation management system to keep track of detected malicious behaviour. We finish with issues important for the functioning of the Sphinx algorithm. In Chapter 4 we present the experiments conducted and their results to prove the correct functioning of the Sphinx algorithm. Finally, in Chapter 5 we will conclude with a summary, conclusions, and recommendations for future work.

Chapter 2

Problem Setting

In this chapter the reader is introduced to the problem setting for this thesis. Nowadays, companies start to deploy their own DHT networks to provide reliable and highly available services to their customers. These DHTs operate in closed environments, owned and controlled by the company. However, history has made it clear that even protected closed networks can be intruded by non-authorised people. Even though this is unlikely to happen, we need to take the possibility into consideration, because when an intruder gets access to a DHT that supports business critical services, he can do serious harm to the functioning of the DHT.
In Section 2.1 we argue why we want to protect corporate DHTs. In Section 2.2 we present an overview of structured overlays. This section is taken from [14]. In Section 2.3 we describe the Chimera protocol on which the Sphinx algorithm is implemented. In Section 2.4 an introduction is given to the environment setting. Here we give a description of corporate networks and their properties and reliability. In Section 2.5, we describe the problem statement of this thesis.

2.1 Why Corporate DHTs?

Besides the clear commercial interests of protecting business critical applications, corporate network environments have the advantage that their properties are better known than the properties of general wide area networks (WAN) like the Internet. Added to that, corporate networks are much more reliable than WANs like the Internet. Therefore, it is harder for a malicious node to shift responsibility for dropped or delayed packets to the underlying network and therefore guilt can be more reliably determined. And, because corporate networks are closed networks, it is unlikely that a large fraction of the nodes get compromised. These conditions make it possible to create an algorithm that can protect DHTs by detecting and locating these few compromised nodes in the network without creating too much overhead, and thereby taking into account the possibility of collusion. Hence, we need to develop an algorithm that is a solution to the following problem: When there is a structured overlay network running in a closed corporate environment

where network properties are better known and more stable than on the Internet, how can we detect and locate (colluding) malicious nodes in an efficient manner?

2.2 Structured Overlays

In this section structured overlay networks are introduced. We give the reader an idea of how these overlays build a decentralised network and how messages are routed in such networks. We start by giving a brief introduction to unstructured overlays and we explain why structured overlays are needed.
Subsequently we discuss the base of every structured overlay and we conclude with a few examples.

2.2.1 Unstructured Overlays

In the early days of P2P networking the pioneers like Napster[1], Gnutella [9] and Freenet [1] proved that decentralised services like file sharing and distributed storage have a huge potential in today's society. However, to let the peers contact each other Napster depended on a centralised server which indexed the files the users had available to share. This dependency on a centralised server made Napster not very scalable and vulnerable for a single point of failure attack (in the case of Napster this was a legal attack). The Gnutella network, which is a typical example of an unstructured overlay network, has no dependencies on centralised services, but it cannot guarantee that a file that is available in the network can be found by every user. The search method for Gnutella is to broadcast a search request to every neighbour in a node's connection table. The neighbours on their turn re-broadcast the search query to their neighbours. Every re-transmission the time-to-live (TTL) value is decreased. If the node that owns the particular file, is not found within the maximum TTL, the search query returns nothing. Besides that, it is clear that broadcasting search queries to every possible neighbour is not very network efficient. Gnutella is an unstructured overlay and cannot even in a fault free network provide guarantees that stored objects will be found. Structured overlays on the other hand allow a node to find another node or object in a deterministic way in a relatively small amount of steps.

2.2.2 Key Based Routing & DHT

The base of every structured overlay is Key-Based-Routing (KBR) [7]. In KBR a message M with destination key K is routed toward the root for key K, where the root is the node that owns key K. The root usually is the node that is *closest* to key K. *Closest* in this case is protocol specific, e.g., it can be the smallest exclusive-or distance between key and node id. A key is owned by just one node at the same time. A key is a n-bit string. Every node identifier is of the same type as a key.

[1] www.napster.com

With every step in the routing path the message gets closer to the destination, i.e., the message is forwarded to a node with an id closer to key K.
A distributed hash table (DHT) implements key-value storage on top of KBR. To store a value with key K, a message like [PUT, value] can be sent to key K. The root then stores the key-value pair and any time it receives a GET message sent to key K it returns the stored value.

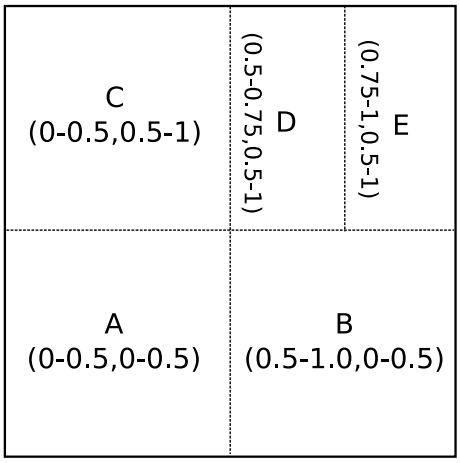

Figure 2.1: Example 2-d coordinate overlay with 5 nodes [8].

Content Addressable Network One of the first structured overlay networks published is the Content Addressable Network (CAN) [8]. CAN is based on a d-dimensional Cartesian Coordinate space on a d-torus. Every key-value pair that needs to be stored is mapped onto a zone by hashing the key to coordinate P. Every zone is owned by a node that stores the keys. Messages are greedily forwarded to the neighbour whose coordinate space is closer to the intended destination. This is achieved by keeping all the neighbours (all zones that share $d-1$ dimensions) addresses. Nodes can join the network by randomly choosing a point P and sending a JOIN message to P. The node responsible for the zone containing P then splits its zone and makes the joining node responsible for half the zone by transferring all the key-value pairs that fall in the new zone. Their own routing tables are updated and messages are sent to the new and old neighbours to make them update their routing tables. The average routing path length for a CAN with n nodes, with zones of equal size (which can be obtained by an uniform hash function on both node IDs and keys) is $(d/4)(n^{1/d})$. Every node maintains routing information about their $2d$ neighbours.

Chord Chord [13] is another structured overlay that, just as CAN, has the disadvantage that it does not attempt to proximate real network distances in its topology construction [16]. Chord creates a ring topology where each node is assigned an

id by hashing a node's IP-address. To create a balanced key distribution, keys are also generated by hashing an identifier, such as a filename. A key k is mapped to the node whose id is the successor of k (in other words, the key is mapped to the first node clockwise after key k).

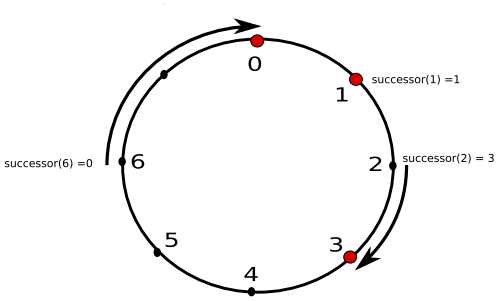

Figure 2.2: An Chord identifier circle consisting of three nodes 0,1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0 [13].

Every node maintains a routing table (called finger table in the original paper) with at most m entries, where m is the maximum number of bits in the key/node identifiers. Hence, a node maintains the addresses of the successor nodes of the keys $n + 2^i$ for every $0 \leq i < m$. A message is routed to key k by routing the message to the node in the table whose id precedes k. This makes the average routing path length $O(\log n)$.
A node join is realised by sending a JOIN message to a bootstrap node containing the new node's id. The bootstrap node finds the m successors to fill up the node's finger table. Finally, the other nodes finger tables are updated (we refer the reader to [13] for the exact algorithm).

Pastry Pastry [10] is an overlay network based on prefix routing. Identifiers are seen as a sequence of digits with base 2^b, where b is a system wide configuration parameter. In every routing step the current node forwards the message to the node f, where node f is the node in its routing table whose id shares a prefix with key k of at least one more digit than the current node shares with the key. Hence, the routing path is less than $\lceil \log_{2^b} N \rceil$ under normal operation, and therefore every node must maintain a routing table with $\lceil \log_{2^b} N \rceil$ rows, each row containing $2^b - 1$ entries. Every entry at row n contains a node that shares a prefix of length n with the present node's id and whose $n + 1$th digit is the column number (that is, one of the $2^b - 1$ possible values other than the $n + 1$th digit of the present node).
For every entry one of many possible nodes can be chosen. In practice a node is chosen that is close in terms of latency to the present node. This provides Pastry with good locality properties in contrast to CAN and Chord. A node joins the network by sending a JOIN message with a key equal to its own id to a bootstrap node. This message will arrive at the node with an id closest to the joining node's

Pastry Node ID 0213			
0	1322	2043	3231
0032	0112	**2**	0330
0201	**1**	0223	0231
	0211		**3**

Table 2.1: The routing table of a hypothetical Pastry node with id 0213, with base 4 and length 4. The bold numbers in each row show the corresponding digit of the current pastry node id.

id. All nodes encountered on this path from the bootstrap node to destination node will send their state tables to the joining node. In this way the joining node can build its own routing table.

Kademlia The Kademlia [5] overlay is one of the many other structured overlays, except that it is one of the few that are extensively used by millions of users [12]. File sharing applications like eMule[2] and aMule[3] based their protocols on Kademlia.

One of the basic principles of Kademlia is that it must be easy to understand. In contrast to many other protocols, Kademlia uses just a single algorithm from the beginning to the end. Protocols like Pastry often use a second algorithm based on numerical difference when a target is almost reached. Kademlia is based on the notion of exclusive or (XOR) distance, i.e., the distance between node x and y is defined as $d(x, y) = x \oplus y$.

For each $1 \leq i < l$, where l is the identifier length, each node keeps a list with pointers to a maximum of k nodes of distance between 2^i and 2^{i+1}. This redundancy is implemented so that every node can start a look-up query in parallel to avoid time delays from failed nodes. These lists are called k-buckets, where k is a system wide configuration parameter. In every k-bucket the nodes are kept sorted by the time last seen (the most recently seen at the tail). When a node receives a message, the sender is placed at the tail. If the sender is not yet in the k-bucket and the bucket is full, the least recently seen node is pinged to check if it is still alive. If it answers, it is placed at the tail of the bucket. If the node does not answer, it is replaced by the new sender.

The Kademlia protocol works in an iterative way, i.e., it queries the closest nodes to a target for even closer nodes. When it receives closer nodes, it queries them again, etc., until it reaches the desired target. The routing table of a node is a binary tree, initially consisting of one node containing one k-bucket. When a new node arrives, it is inserted in the k-bucket. When the k-bucket is full, the bucket is split. It is shown that most operations take $\lceil logn \rceil + c$ time [5], where c is a small constant.

[2] www.emule.org
[3] www.amule.org

2.3 Chimera

This section describes the Chimera protocol. We use this protocol to implement and test the additions we will design to detect and locate malicious nodes in a structured overlay network. Chimera is an implementation in the language c, of the routing and location parts of the Tapestry protocol [16] developed at the University of California Santa Barbara (UCSB). Tapestry is an overlay infrastructure designed to enable the creation of scalable, fault-tolerant applications in dynamic wide area networks.

2.3.1 Fault Tolerant Routing

Tapestry is similar to Pastry. It uses prefix routing and similar insertion/deletion algorithms. There are several key differences between Tapestry and Pastry in the way objects are stored and duplicated. We will not elaborate on this, since it is not relevant because Chimera only implements the routing part of Tapestry.
Besides a routing table, every Chimera node keeps two leaf sets, which are lists of the L closest nodes the node is aware of at each side, where L is a predefined constant. Hence, the left leaf set contains the $L/2$ closest nodes with a key lower than its own. And, the right leaf set contains the $L/2$ closest nodes higher than its own. The leaf sets are used to directly forward a message to the node with the smallest numerical distance to the destination key.
To realise fault-tolerant routing, Chimera nodes keep their routing tables reliable by sending periodic ping messages to the nodes in their routing table and leaf set. For every node in the routing table and leaf set a success rate is kept. The success rate is calculated by taking the average success of the last M messages sent to a node, where M is a predefined constant usually set to 20. Success is defined by whether an acknowledgement of a message for a node is received within a predefined period of time or not. Each entry in the routing table contains two backup nodes in with the same shared prefix. On failure these nodes can be used.
Chimera implements surrogate routing to find a root node for a particular key. To find the destination node for a key, messages are forwarded to the node with the same id as that key as if such a node exists. When a node cannot route the message to a closer node, it is the destination node.

2.3.2 Node insertion

When a new node n wants to join the Chimera network, it first generates its own id key by SHA-1 hashing the concatenation of its host name and listening port. The new node must be aware of a bootstrap node it can connect to. After connecting to bootstrap node G, node n attempts to route a JOIN message. As the destination of the JOIN message, it uses its own id key. The JOIN message will finally be received by the node with the closest id to the id of node n. When the JOIN message traverses the routing path, every node on the routing path sends the row

from its routing table that matches the longest prefix of the id of n to n. With every step the common prefix of the key of the current node with the key of node n becomes one position larger. Hence, with every step node n gets a new row for its routing table. After l steps node n should have received a complete routing table, where is l is the predefined length of a key.

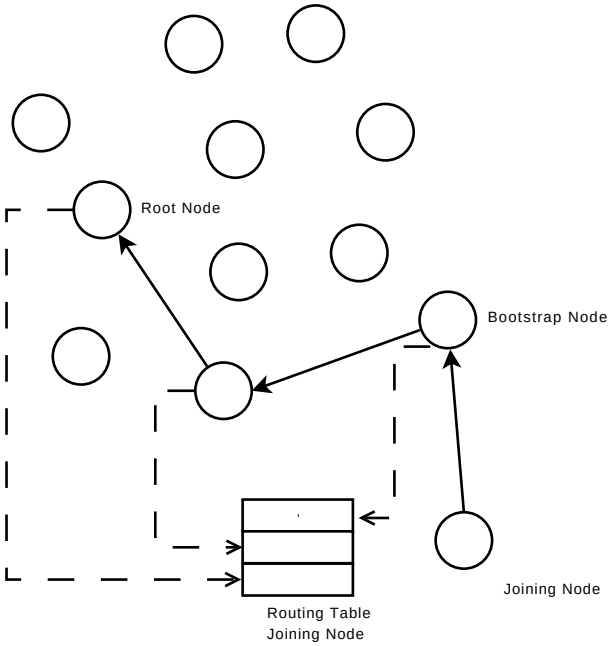

Figure 2.3: Joining node sends JOIN message to current root node of own key.

When the message arrives at the current root node of the id of node n, it replies with a JOINACK message and sends its leaf set to n. When node n receives the JOINACK message, it updates its leaf set and announces its arrival to all nodes in its routing table and leaf set by sending them an UPDATE message. The neighbours update their routing tables. Node n has joined.

2.4 Corporate Overlay Networks

In this section we introduce the reader to the context of our problem. The environments in which our problem occurs are corporate overlay networks, i.e., overlay networks running in a corporate WAN. Corporate networks usually consist of fully controlled data-centres, which are connected with leased data lines with a guaranteed available bandwidth.

2.4.1 Properties of Corporate Networks

A corporate overlay network is an overlay network within the boundaries of a corporate WAN. A corporate WAN has multiple advantages over a WAN like the Internet. Where for the Internet there are no guarantees at all about transmission times, latency and reception, there are some assumptions we can make about corporate WANs. First, a corporate WAN typically consists of multiple data centres connected with leased lines. Therefore, realistic estimates can be made about latency and transmission times. Whenever the actual numbers deviate from the estimates, it is possible to blame nodes with much more certainty. Hence, if we deploy an overlay network in such an environment, we can detect and locate malicious behaviour of participating nodes. Second, corporate WANs are usually closed systems. Therefore, it is unlikely that malicious code is run in the network. However, we should still keep in mind that it is possible that nodes get compromised by an attacker. Hence, it is unlikely that malicious nodes reside in the network, but we should seriously consider the possibility and whenever it happens we should detect and locate the nodes before serious damage can be done to the network.

2.4.2 Dynamo

An example of a structured overlay network running in a corporate environment is Amazon's highly available key-value storage system called Dynamo [2]. Dynamo is a highly available and scalable distributed data store built for Amazon's platform. It is a completely decentralised system where nodes can be added and removed without requiring any manual partitioning or redistribution. Dynamo is used to store the state of services that have very high reliability requirements and need tight control over the trade offs between availability, consistency, cost-effectiveness and performance. To meet the stringent latency requirements that 99.9% of all read and write operations are performed within a few hundred milliseconds, the designers wanted to avoid routing requests through multiple nodes. Hence, each node maintains enough routing information to route a request to an appropriate node directly.

Many services on Amazon's platform that need these properties, such as those that provide shopping carts and best seller lists only need primary key access. A relational database cannot provide the required properties mentioned. Every service that uses Dynamo runs its own Dynamo instances.

Dynamo uses consistent hashing [4] to partition the data uniformly. In consistent hashing a ring is created by wrapping the largest hash value around the smallest hash value. Objects are stored under a certain key on the first node with a larger identifier than that key. To prevent heterogeneity in the performance of nodes, Dynamo uses the concept of "virtual nodes", i.e., every node has multiple identities uniformly distributed over the ring. Hence, when a node becomes unavailable the load handled by this node is evenly dispersed across the remaining nodes.

Dynamo uses replication to achieve high availability and reliability. Data is repli-

cated over N nodes. A node responsible for a certain key K stores an object locally and on its $N-1$ clockwise successor nodes. To prevent objects to be stored on virtual nodes belonging to the same physical node some positions in the ring are skipped. Partitioning and placement information propagate via a gossip-based protocol. Hence, each node can forward a key's read/write operation directly to the correct node.
The authors assume that most node failures are temporary and should not result in re-balancing of the partion assignment. Hence, the authors chose to use an explicit mechanism to add or remove nodes. For failure detection Dynamo uses a purely local detection mechanism. When node A does not get a response from node B, node A considers node B failed, even if node C can still reach node B. Node A now uses alternative nodes to service requests that map to B's partition. Periodically node A retries B to check for a recovery.

2.5 Problem statement

DHT networks within a corporate environment have certain advantages over DHT networks in an open environment. First, the nodes in the network are all controlled, i.e., the network consists of trusted nodes. However, even nodes in a closed network can be attacked by an intruder. When a non-authorised individual succeeds in taking over a node or creating nodes he controls, within the closed DHT, these nodes are able to disrupt the functioning of the DHT. Especially when such a DHT is used in a production environment and needs to be highly available and reliable, this can have disastrous consequences.
In this thesis we present the design and implementation of the Sphinx algorithm that protects DHTs when intruders get access to the network. In particular, Sphinx will have the following capabilities:

1. When an intruder creates new nodes especially designed to disrupt the correct functioning of the network, it needs these nodes to join the network. Our algorithm prevents the joining of any unauthorised node.

2. An intruder that is able to take over a node can replace the compromised node with a malicious node with the same identity. This malicious node can disrupt the correct functioning of the DHT by dropping messages it should have forwarded. A malicious node may also deliberately delay messages, change messages, or route messages the wrong way. The Sphinx algorithm detects this malicious behaviour and locates the source of this malicious behaviour. In summary, the Sphinx algorithm must be able to detect and locate the following malicious behaviour:

    - Dropping of messages
    - Delaying of messages
    - Misrouting of message, i.e., forwarding messages to the wrong node.

- Changing the payload of messages

Hence, the Sphinx algorithm must be able to detect malicious behaviour in an effective and efficient way. When this behaviour is detected, the Sphinx algorithm must be able to locate the node that causes this malicious behaviour. This detecting and locating must be achieved within a reasonable amount of time, while minimising overhead costs. When there is a trade off between detection speed and overhead minimisation, the emphasis in Sphinx is on overhead minimisation.

Chapter 3

The Sphinx algorithm

In this chapter we present the Sphinx algorithm for detecting and locating malicious nodes in a structured overlay network running in a corporate WAN in a scalable way. In Section 3.1 we present an overview of how the Sphinx algorithm works. In Section 3.2 we describe how and why signing acknowledgements is the foundation of the Sphinx algorithm. In Section 3.3 the basic operation of how malicious nodes are detected and located by the Sphinx algorithm is explained. In Section 3.4 we discuss how we manage the reputations of nodes in the network and keep track of malicious behaviour. In Section 3.5 a few remaining issues, like increasing signing speed, time synchronisation and smearing of nodes are discussed, and the solutions Sphinx uses for these issues are presented.

3.1 Overview

In order to detect malicious behaviour in message passing, and locate malicious nodes in a DHT network in an effective and efficient way, the Sphinx algorithm relies on the presumption that every node in the network must be able to prove its innocence when malicious behaviour is detected. When a message is transferred along a path of nodes, every node must be able to prove that it forwarded the message correctly to the following node on a path. The Sphinx algorithm realises this by letting every node request a timestamped signed acknowledgement from the node it forwarded a message to. When malicious behaviour is detected, this signed acknowledgement can function as a “proof of innocence”, i.e., when an investigator (that is the node that wants to locate the malicious node) wants to locate the malicious nodes it iterates through the path starting at the first hop it sent the message to, repeatedly requesting a “proof of innocence”. When a node cannot present a signed acknowledgement, or only a timestamped signed acknowledgement which does not correspond to the expected properties of the corporate WAN it can be considered malicious.

3.2 Signed Acknowledgements

We now give a detailed description of the Sphinx algorithm for detecting and locating malicious nodes in a structured overlay network.
The base of the Sphinx algorithm is a public-key infrastructure (PKI). The PKI is used for two purposes. First, the PKI is used to prevent unauthorised nodes from joining the Sphinx DHT. Second, the PKI is used to let nodes send each other signed acknowledgements, i.e., acknowledgements containing a signature that proves the acknowledgement was created by the sender (non-repudiation).
One part of the PKI is a centralised trusted Certificate Authority (CA). Every node that wants to join the Sphinx network must possess a digital certificate handed out by a CA. This certificate contains the following information:

- The key of the joining node, which is generated by the CA. The CA uses a secure hash function like SHA-1 to generate uniformly distributed keys.
- The IP address and port of the joining node. This prevents the re-usage of a certificate by a different node.
- The public key of the node. The private part of the public/private key pair will be needed by the joining node to sign messages. With the public key, other nodes are able to verify the validity of messages that are signed by the joining node.

The certificate is signed by the CA. Every node in the Sphinx network is in possession of the public key of the CA and therefore is able to verify if a certificate is valid.
When a node wants to join the network, it must present the certificate it received from the CA. Every node that wants to add the joining node to its routing tables should verify the validity of this certificate.
Certificates are manually created by the operator of the network when a new node joins the network. By using a PKI, an intruder will not be able to create new nodes to join the network because it cannot create its own certificates. However, an attacker that is able to take control over an already running node will be able to use the identity of that node. With this stolen identity the attacker can create a node that disrupts the functioning of the DHT.
The heart of the Sphinx algorithm is that every node must be able to prove it is behaving correctly. When a node is not able to do so, it is considered malicious.
To be able to prove that a node is not malicious the node expects a signed acknowledgement for every message it sends or forwards from its successor node in the message path. Therefore, the receiver of a message creates, signs and sends an acknowledgement. Table 3.1 shows a Sphinx acknowledgement structure. The acknowledgement contains:

- Sequence # and source of the message, to uniquely identify each message.

Sequence #
Key of Source of Message
Receive Time
MD5 hash
Key of Signing Node
RSA Signature

Table 3.1: Acknowledgement Structure.

- A time-stamp containing the receive time, to detect forward latency.
- A hash of the content, to prove the message's payload is not modified.
- The identity key of the acknowledging node, to show who is the next responsible node and to detect misrouting.
- The RSA signature, to ensure non-repudiation.

This process is shown in Figure 3.1

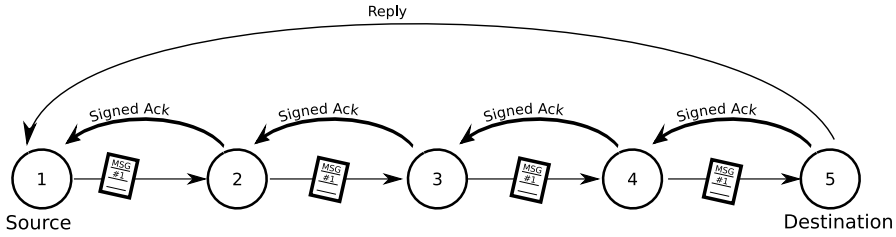

Figure 3.1: The operation of the Sphinx algorithm: A message is forwarded from source to destination. Every node in the path acknowledges the message to its predecessor in the path.

With this signed acknowledgement a node can prove it forwarded the message correctly and on time to the node that acknowledged the message, to whoever is interested in questioning the node's innocence. In other words, this signed acknowledgement is its "proof of innocence" for this particular message. Therefore a node must check if all these 5 properties of the signed acknowledgement are correct. In short, the only thing a node should worry about is getting this "proof of innocence" for every message it forwarded.
If the receiving node does not acknowledge a message, the node where the message is currently residing has the responsibility to send the message to a different neighbour. The only concern a node should have is to receive a signed acknowledgement that relieves him from the burden of being responsible for that message. Therefore we presume that a node has alternative nodes in its routing table to forward a message to, which is the case in every DHT.
A few conditions must be met for the Sphinx algorithm to work:

1. A node must not have the ability to create its own identities. If a node can create identities, it can reenter the network with a different id. Since a node needs a certificate issued by the CA this is not possible as long as the CA can be trusted.

2. When a node encounters a node that does not return correct acknowledgements, it is important that it stops forwarding messages to that particular node. It should therefore remove the node from its routing tables.

3. Even in a corporate environment, the network is not 100% reliable. Therefore, when a node seems non-cooperative, it does not necessarily mean that a node is malicious, even though it is likely. Hence, we need an algorithm to distinguish malicious behaviour from network malfunctioning.

3.2.1 Malicious Behaviour

When a malicious node resides in a Sphinx DHT and it wants to do harm, its options are now limited, because malicious behaviour will be detected. It has a number of strategies to disrupt the functioning of the DHT:

1. Not taking responsibility for a message by not returning a signed acknowledgement. With this strategy however, the malicious node will be quickly detected and removed from its neighbours routing tables.

2. Another strategy is to accept its responsibility by returning an acknowledgement but after that dropping or delaying the message. This strategy however, is even worse, because now the node can be blamed but it cannot prove its innocence.

3. A third strategy is claiming that it is the root node. Also referred to as identity theft. For this strategy we could use the solution proposed by Ganesh et al. [3], i.e., using “proof managers” and proof of existence. For more information we refer the reader to the article.

3.3 Detecting Malicious Nodes

In the Sphinx algorithm it is always the source node of a message that detects malicious behaviour and locates the malicious node. There are two reasons to start locating a malicious node:

1. First, when a source node does not receive an expected reply from the destination within a certain period of time or at all, it can suspect that somewhere in the path from the source to the destination a malicious node resides. However, waiting for a reply from the destination requires that either the sent message causes a reply (like a read operation) or the algorithm always acknowledges every message to the source node.

2. The second method, is to randomly start the locating algorithm for every L-th message, where L is a system wide parameter. The advantage of this method of random checks is that not only malicious behaviour can be detected and located, but path statistics can be obtained as well. These statistics can be used to monitor the performance of the network in general. For instance, to calculate expected values for latency times.

When the source node suspects malicious behaviour for a certain message somewhere in the message path, using one of the two methods described, it starts querying the first node in the path the message was sent to, since it is the only node in the message path it knows about. This first node responds by presenting the signed acknowledgement it received from its successor, to the source. This signed acknowledgement contains the identity of the successor, the MD5 hash of the message and the receive time. By comparing the MD5 hash to the hash of the original message, the source can detect if the message's payload has not been modified.

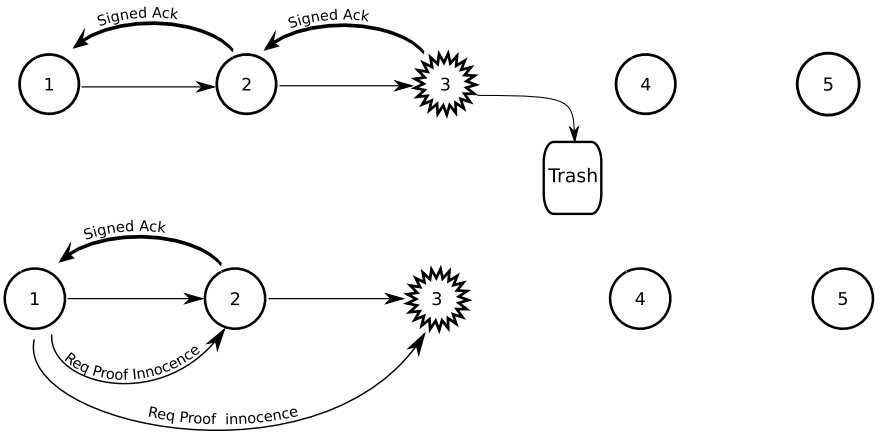

Figure 3.2: A message is dropped by node 3. The source node queries for signed acknowledgements. Node 3 cannot present a signed acknowledgement and is considered malicious.

By comparing the receive time of the successor of the first node in the path to the receive time in the acknowledgement received by the source node, the source node can calculate the forward time. If the forward time or the MD5 hash do not comply with the expectations, the first node in the path may be malicious. Or at least it displayed malicious-like behaviour. If malicious behaviour is not detected, the source node queries the next node in the path until the source reaches a node that cannot present a signed acknowledgement which proves its innocence. Now the node causing the anomaly, is located.

3.4 Reputation Management

When malicious behaviour is detected, we need a way to store this information to calculate the reputation of a particular node. The storage of this information needs to be done in a scalable way. The reputation r of a node is represented as a number between 0 and 1, and is equal to the probability that a non-malicious node would display the behaviour the particular node displays. Hence, when r is close to 0, a node is likely to be malicious, and when r is close to 1, a node is unlikely to be malicious.

3.4.1 Reputation Managers

To store reputations in a scalable way, every node in the Sphinx DHT has 3 other nodes that keep track of its reputation. We call these nodes "reputation managers". Hence, reputation managers are nodes that store and calculate the reputations of other nodes in the network. Sphinx uses three reputation managers for each node to prevent a malicious reputation manager from propagating wrong reputations. When a reputation for a node needs to be retrieved, the three reputation managers are queried, and the two reputations closest to each other are averaged. To determine which three nodes in the network are the reputation managers for a certain node n, the key of node n is hashed three times by using SHA-1, each time concatenated with a different salt: 0, 1 or 2. The reputation managers are the root nodes for these three keys. Hence, each node has three different reputation managers, which are uniformly distributed over the DHT. Therefore, on average, every node is a reputation manager for three other nodes in the network. This also implies that the reputation storage and calculation load is uniformly distributed over the DHT, so, this mechanism is scalable.

3.4.2 The Blame Process

Figure 3.3 shows how a Sphinx node acts when malicious behaviour is detected: When the source node n_1 detects malicious behaviour somewhere in the message path, it starts to locate the node that caused this anomaly as described in Section 3.3. When node n_2 is identified as the node that caused the malicious behaviour, node n_1 sends a BLAME message to the three reputation managers of node n_2. If a latency violation is detected, the BLAME message contains two signed acknowledgements to prove the latency deviation: The acknowledgement received by the predecessor node of the blamed node, in this case, node n_1, and the acknowledgement received by the blamed node. In the case that node n_2 did not forward message 1, and therefore can not provide a signed acknowledgement from node n_3, only one acknowledgement is sent along with the BLAME message. However, now the reputation managers should also request a signed acknowledgement from the accused node to verify this claim.

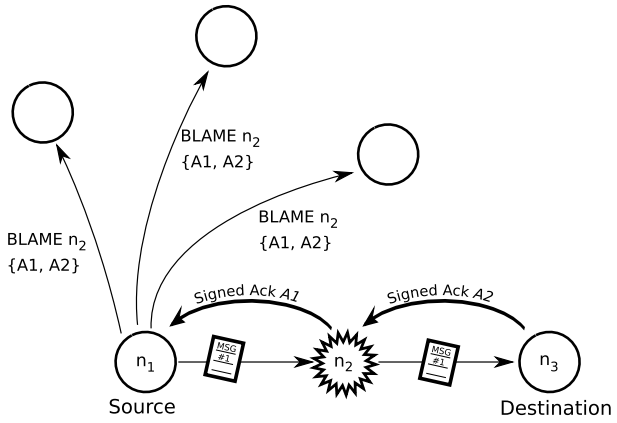

Figure 3.3: The blaming of malicious node n_2 by node n_1. Blame messages are sent to the reputation managers of node n_2.

The reputation managers now store the BLAME message and calculate the new reputation of the blamed node.

3.4.3 The Reputation Calculation

The reputation calculation consists of three parts:

- Verification of the accusation
- Querying for messages sent by blamed nodes
- Calculation of the new reputation

Verification

When a reputation manager receives a BLAME message for a certain node, it first checks if it did not receive a BLAME message for the same node within the last T seconds, where T is the system wide time out parameter. Because a node needs T seconds to find out whether a message is not acknowledged on time, it could have forwarded more messages to the same non-responsive node in the mean time. Hence, if the reputation manager did receive an accusation the last T seconds for this node, it drops the current accusation.
Second, the reputation manager checks whether the accusation is genuine. If the node is accused of latency violation, the accusation should be accompanied by two signed acknowledgements. If the node is accused of dropping messages, the accusation should be accompanied by one signed acknowledgement. To verify that the accused node indeed dropped the message, the reputation manager should request the second signed acknowledgement from the accused node. If the accused node cannot provide the second signed acknowledgement (its proof of innocence), then the accusation is valid.

Querying

To calculate the probability of a node being malicious, one should know the number of messages forwarded by the node, besides the number of latency violations. However, the number of messages n forwarded by the blamed node, is unknown by the reputation manager. Hence, the blamed node needs to be queried for this number. Taken into account that a malicious node has every reason to exaggerate this number, we need a non-forgeable way to retrieve this number.
The easiest way to query this number is by requesting the signed acknowledgements for every message the node forwarded. However, this induces a large network traffic overhead when n is high. E.g, if a node forwarded 10^4 messages, this means it should transmit $10^4 \cdot 134 = 1.34$ MB in the worst case, where every single message is acknowledged separately. And 284 KB in the best case, where all messages sent are acknowledged in groups. See Section 3.5.5 for details. All acknowledgements also need to be sent to each of the three reputation managers. This is unfeasible, and therefore, we need to use a random check method.
For a random check method however, we need claims we actually can verify. Hence, a blamed node should give all the identities of the messages it forwarded. A message can be uniquely identified by its source and its sequence number, respectively 20 and 4 bytes. After receiving the message identifiers, the reputation managers can query multiple signed acknowledgements for randomly chosen messages. In case 10^4 messages have been forwarded, the total data to transmit now equals $10^4 \cdot 24 = 240$KB, plus the transmitted data needed for the requests and the returned acknowledgements. This is a major improvement, but still a lot of overhead. To further reduce the overhead, we use the following method we called "divide and expose'. This method directs the blamed node to a certain timeframe where the node finally should be able to present a valid signed acknowledgement.

1. Reputation manager r queries blamed node b for number of messages forwarded n, since the last blame message received for node b.
2. Node b responds with a number n and a timestamp t that divides the forwarded messages in two equal parts.
3. Reputation manager r randomly selects one half to research further. It now requests node b to return a timestamp that divides the messages in the selected half into two.
4. The reputation manager repeats step 2 and 3 $\lceil log_2 n \rceil$ times in total.
5. Finally, when one message is singled out, the accompanying acknowledgement is requested. If node b cannot present a signed acknowledgement for a message sent in the last selected time frame, it is clear node b is malicious.

This method works, because the blamed node does not know which half will be selected by the reputation manager on each iteration. Hence, it cannot direct the querying to a given outcome.

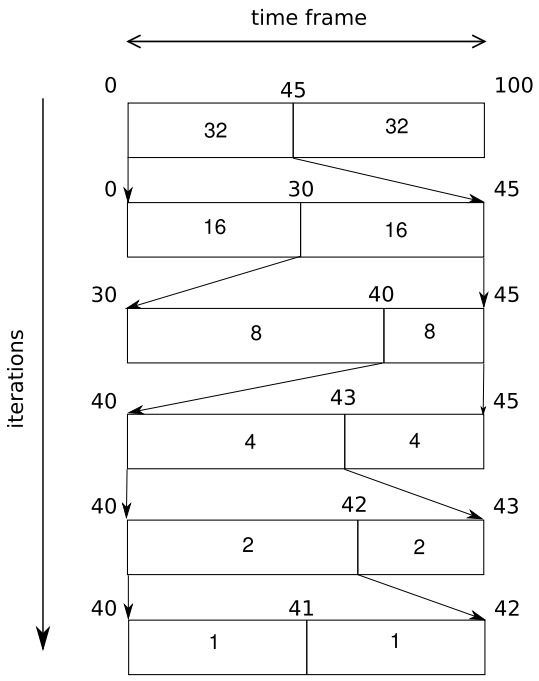

Figure 3.4: An example of a non-parallelised “divide and expose” query in which the blamed node sent 64 messages in period [0,100].

The probability of catching an exaggerating node is $1 - (n/n')$, where n' is the exaggerated value and n the real value. To increase the probability that a malicious node gets caught when exaggerating the number of messages it has forwarded, we choose to use multiple “divide and expose” sessions in parallel.
The overhead generated by this method is calculated as follows: A timestamp is an 8 byte floating point number, and to indicate which half a reputation manager wants to research further 1 bit is enough. Hence the total transmission overhead generated is $\lceil log_2 n \rceil \cdot (8p + \lceil p/8 \rceil)$, where p is the number of parallel sessions. For instance, when $n = 10^4$ and $p = 8$, the total overhead generated is as low as 910 bytes, which is more than 240 times less than the normal “random check” method. Figure 3.4 shows an example of non-parallelised “divide and expose” query.
We considered the fact that a malicious node can always increase n by sending garbage messages to its neighbours, and therefore receiving signed acknowledgements. However, checking if sent messages are useful or not should be handled in the application layer, so, we will not elaborate on that.

Calculation

After verification of the BLAME message, the reputation manager stores the BLAME message accompanied by the signed acknowledgements, and updates the current reputation of the accused node. As mentioned above, the reputation equals the

Number of messages sent	Number of violations allowed
100	4
1000	10
10000	30
100000	156
1000000	1169

Table 3.2: Number of latency violations allowed for different values of the number of messages sent before reaching the reputation threshold.

probability that a node displays its current behaviour without being malicious. Whenever this probability drops below a certain threshold where it is very unlikely that a non-malicious node would display this behaviour, the node is considered malicious. The Sphinx algorithm uses 10^{-7} as this threshold. Hence, that is the probability a node gets the false reputation of being malicious. Once a node is considered malicious there is no way to increase its reputation, and therefore, it stays malicious.
The probability that a node displays the behaviour it does, is calculated as follows. The Sphinx algorithm considers the maximum allowed latency as the 99.9th percentile of the latencies measured for the current network. Hence, the probability of latency violation without maliciousness is $p = 10^{-3}$. The probability r_n that a latency violation will happen at least k times after sending n messages is given by:

$$r_n = \sum_{i=k}^{n} \binom{n}{i} p^i (1-p)^{n-i} \tag{3.1}$$

To be sensitive, the period of time that is taken into account should be small. However, a small period of time allows a malicious node to continuously cause the maximum number of latency violations it can afford (reputation-wise). Therefore, the Sphinx algorithm calculates the probability r_n of Eq. 3.1 over different periods of time: respectively, for the last 10^2, 10^3 , 10^4, 10^5, and 10^6 messages sent. A reputation manager can do this because it knows the number of messages sent between successive BLAME message it received for a node. The reputation will equal the lowest of the five values of r_n calculated above.
Table 3.2 shows how many latency violations are allowed for different values of n before the threshold of 10^{-7} is reached.

3.5 Issues

In this section we describe the issues that are important for the correct functioning of the Sphinx algorithm. In Section 3.5.1 the problem of colluding nodes in discussed. In Section 3.5.2 we discuss the timing issues caused by the forward speed enforcement functionality. In Section 3.5.3 we present two methods to overcome

the problem of signing being compute intensive. Section 3.5.4 we describes how Sphinx handles smearing of nodes. In Section 3.5.5 we give a detailed description of the overhead generated by the Sphinx algorithm.

3.5.1 Colluding Nodes

When an intruder manages to take over multiple nodes, he could set up these nodes to disrupt the functioning of the DHT by colluding.
Traditional colluding attacks, like the eclipse attack [11], are not possible because the CA uniformly distributed identities between the nodes. The only attack feasible for colluding nodes as shown in Figure 3.5 is presenting genuine signed acknowledgements to each other, i.e., one malicious node protects another malicious node by taking over its responsibility.
However, by taking over the responsibility from another node, the last node in the colluding group will be blamed for the malicious behaviour. Hence, a colluding group of nodes can only disturb the operation of the structured overlay for as long as not all colluding nodes are detected and located. If the group consists of a large number of malicious nodes taking over responsibility from each other, or nodes possess a large number of identities, that makes sense. However, because identities have to be created by a certificate authority there are not an unlimited number of identities available. An intruder can only join the DHT with the identities of nodes that have been taken over.
Of course, a researching node must detect loops in the messsage path to prevent malicious nodes from delaying a message by sending eachother signed acknowledgements.

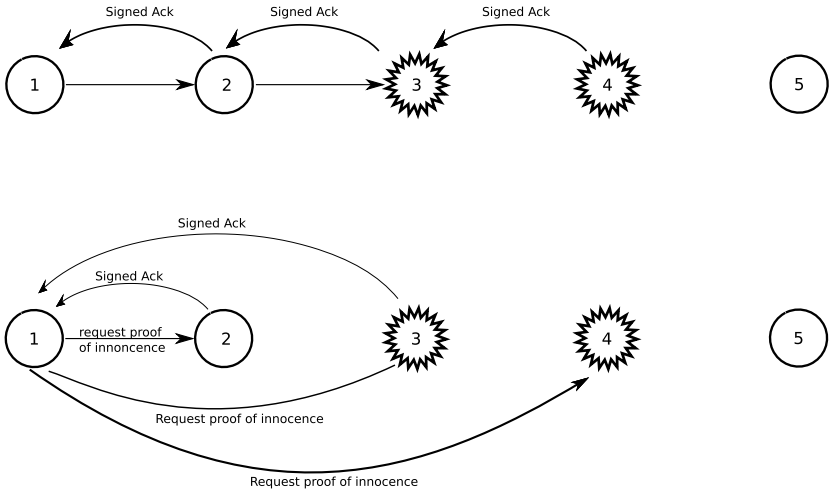

Figure 3.5: Node 4 backs node 3 by providing signed ack. Node 4 can only do so by sacrificing himself.

Besides the problem for the colluding group that always one node will be blamed,

each malicious node in the group should also take into account that it can only forward messages to a node closer to the destination key. Hence, on average only half of the colluding group can be used, because the identities of the nodes are uniformly distributed.

3.5.2 Timing

Because Sphinx does not only require nodes to take their responsibility by forwarding messages, but also by doing so within a certain period of time, we add a timestamp to the signed acknowledgement. For this purpose a receiving node adds the time it received the message from its predecessor to the acknowledgement. The forwarding node checks if this time-stamp relieves itself from the burden of bearing the responsibility for the current message, i.e., if a node can prove it forwarded the message within the defined maximum period of time. Hence, if a received acknowledgement does not relieve the forwarder it must make sure it sends the message to an alternative node.
To be able to implement "forward speed enforcement", there must be global time synchronisation. The Network Time Protocol (NTP) [6] is an excellent way to achieve global time synchronisation. NTP can reach an accuracy of several milliseconds [6]. Hence, a node can substantiate its forwarding speed with that given accuracy.
A few problems arise when using "forward speed enforcement". First, if a node forwards a message to a malicious node who does not acknowledge on time or at all, it must re-forward the message to an alternative node. However, the time it waited for the first acknowledgement is lost and the node may now have violated the enforced forwarding latency itself. Therefore, we need some sort of reputation system, that take these cases into account. Section 3.4 described how Sphinx implements a reputation system for cases like these. In short, forwarding delays are accepted if it does not happen too often. And, if a node causes multiple forwarding delays within a short period of time, only for the first delay the node will be blamed

3.5.3 Signing

One of the practical problems we encounter with the Sphinx algorithm is that signing messages is compute intensive, and therefore we may not be able to reach a high throughput. The Sphinx algorithm implements a number of solutions to decrease the needed computation, and therefore increasing the throughput:

1. *Signing groups of messages.* Instead of acknowledging every single message received from a preceding node in a message path, the Sphinx algorithm acknowledges multiple messages at the same time. Every N messages or every T seconds, all messages received from the same node and with the same source are acknowledged at once, where N and T are system wide parameters. By acknowledging groups of messages at the same time less signatures have to be created.

We can increase the efficiency by grouping these messages by their source. When the created signed acknowledgement is requested by the source node for detection purposes as described in Section 3.3, it immediately has a complete group of relevant acknowledgements. The same principle applies when a signed acknowledgement is used to acknowledge the reception of messages to the source by the destination. And last, as described in the next item, when acknowledgements are forwarded it is very likely that messages received with the same source, traversed the same path. Hence, it decreases the total number of acknowledgements that needs to be forwarded.

2. *Forwarding signed acknowledgements* as shown in Figure 3.6. Because messages are not acknowledged immediately, but only every T seconds it is very likely that at the moment a node wants to create an acknowledgement, it already received a signed acknowledgement from its successor node in the path. When this indeed happened, the node can forward this acknowledgement instead of creating a new one. For the receiving node, it does not matter if it can present a signed acknowledgement from its direct successor or from a node further in the path, because with this forwarded acknowledgement it can also prove it forwarded the message. With this method the total signing of messages can be reduced, and therefore the maximum throughput can be increased.

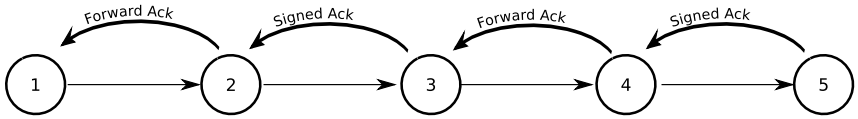

Figure 3.6: Signed acknowledgements are created by nodes 3 and 5, and forwarded by nodes 2 and 4.

Signing groups of messages has the disadvantage that a node receives its "proof of innocence" only after T seconds or N messages. Hence, malicious nodes can corrupt N messages before being detected. However, the period of time in which detection and location of a malicious node is realised may be larger, but the amount of computation needed can be drastically reduced. And it definitely complies with the design goal of detecting malicious nodes in a reasonable amount of time and with a reasonable amount of overhead.
When a node receives a forwarded signed acknowledgement, i.e., the acknowledgement is not signed by the host it sent the message to, but by another host further down the path, it possesses an acknowledgement that includes receive times of the signing node, and not the receive times of its successor node in the message path. Therefore, the forwarding latency the node can prove will logically increase, because the message traversed multiple hops within that time period.
For example, when node 1 in Figure 3.6 wants to detect the forwarding latency of node 3, it compares the timestamps in the signed acknowledgements received

by node 2 and node 3. Node 3 however, possesses the acknowledgement node 5 created. Hence, the difference between the timestamps indicates the latency time for two hops.

Therefore, when a node forwards a signed acknowledgement it should take into account that the receiving node will not violate the forwarding latency maximum. However, this can only be done when a node is aware of the receiving time of its predecessor node. Therefore, the Sphinx algorithm adds a timestamp to each message sent, that indicates when the sending node received the message itself. Now, a node checks if forwarding a signed acknowledgement will cause a latency violation for its predecessor node. If it would, then the node signs an acknowledgement itself, instead of forwarding the signed acknowledgement. This policy will not influence the forwarding of signed acknowledgements most of the time, because in the Sphinx algorithm, we set the maximum latency to the 99.9th percentile of the measured latencies of the network.

3.5.4 Smearing of nodes

In the Sphinx network there is only one way for a malicious node to *smear* a non-malicious node, i.e, to make a non-malicious node look malicious, or more precisely, to make sure a non-malicious node will be saddled with an invalid signed acknowledgement. Because, when a malicious node can induce such a situation, the non-malicious node cannot prove its innocence.

When a malicious node does not return a (valid) signed acknowledgement to a non-malicious node, the non-malicious should resend the message to a different node. However, now the non-malicious node will possess a signed acknowledgement that indicates a latency violation. This situation can occur, because signed acknowledgements are not returned immediately.

By the time the non-malicious node detects the maliciousness of its neighbour, it is likely that it already sent more messages to exactly that same neighbour. Hence, a non-malicious node may not be able to present correct acknowledgements for a period of time T, where T is the system wide time out parameter. Therefore, when a node cannot present a correctly timestamped signed acknowledgement for message M, it cannot be blamed for the other messages forwarded to the same node within time period T after the node forwarded M. But only if it can present a signed acknowledgement at all. So, when a node does not receive a correct signed acknowledgement, it must immediately resend all messages sent to a malicious host to an alternative host.

The downside of this smearing prevention is that it gives the malicious nodes the ability to delay messages in a period of T for a maximum of T seconds. To still satisfy our design properties and to prevent malicious nodes from continuously delaying traffic, the Sphinx algorithm implements a reputation management policy, as described in Section 3.4

3.5.5 Overhead and Memory Usage

The overhead of the Sphinx algorithm in addition to the Chimera algorithm depends on several system wide parameters, such as the time and number of messages N after which signed acknowledgements are sent, and the average size of the messages sent.
The Sphinx overhead consists of two parts. First, the overhead generated because every node needs to send an acknowledgement to its predecessor node. Second, the overhead generated for the detection of malicious behaviour. There are two detection methods: either the destination node acknowledges the source node or the source node randomly starts the location procedure. Both methods are described in Section 3.3.
For the first part, we need to know the exact structure of the signed acknowledgement, to determine the generated overhead. Table 3.3 shows the structure of a signed acknowledgement. The bottom four fields only appear once in a signed acknowledgements. The greyed row indicates a record that will be included in the acknowledgement for each message the signed acknowledgement acknowledges.
Hence the overhead expressed as a percentage of the message size S can be calculated as

$$\frac{(106 + 28N)/N}{S}.$$

⋮		
Sequence # - 4 byte	Receive Time - 8 byte	MD5 hash - 16 byte
Number of messages acknowledged (2 bytes)		
Key of Source of Message (20 byte)		
Key of Signing Node (20 byte)		
512 bit RSA Signature of Data (64 byte)		

Table 3.3: Acknowledge Message Structure.

For the second part, the overhead generated by monitoring the network for malicious nodes with the different detection methods as is as follows:

- When the destination acknowledges every message to the source node, the destination sends the same acknowledgement it sent to its predecessor node, to the source node. Hence, here the overhead also depends on the number of messages covered by one acknowledgement.

  The overhead also depends on the path length P. Hence, the extra overhead generated by this method expressed in bytes is

  $$\frac{(106 + 28N)/N}{P}.$$

- Using random checks: Using this method, randomly every L-th message is checked for correct delivery. Every node in the path is queried for a signed

acknowledgement. The requests is a 4 bytes message sequence number, and the reply is a signed acknowledgement message as in Table 3.3. The overhead expressed in bytes per message sent is

$$\frac{4 + (106 + 28)}{L} = \frac{138}{L}.$$

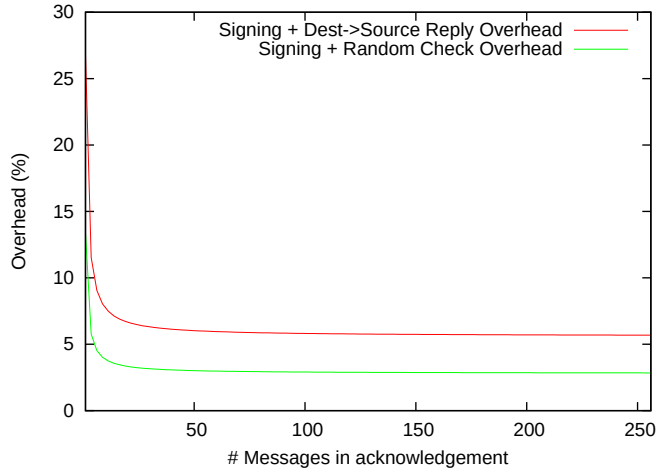

Figure 3.7: Overhead for different detection methods (based on a message size of 1000 bytes).

Figure 3.7 shows the total overhead by using one of the methods, with an average message size S of 1000 byte and a variable number N of messages covered in an acknowledgement.
A Sphinx node has to store all the signed acknowledgements for the messages it forwarded since the last query it received from its reputation managers. Hence, on average it has to store 1000 acknowledgements. It is very unlikely it ever has to store more than 15000 acknowledgements, because the probability that it does not get blamed after sending 15000 messages equals 10^{-7}.

3.6 Implementation Details

This section describes some of the implementation details of Sphinx.
A sent message contains the following data:

1. The Message Type: An integer containing the type of the message, clients using the chimera library can register callback functions for each message type.

2. The Destination Key: A destination key to route the message to.

Payload
Receive Time - 8 bytes
Secure Flag - 1 byte
Sequence Number - 4 bytes
Source Key - 20 bytes
Destination Key - 20 bytes
Size of Payload - 4 bytes
Message Type - 2 bytes

Table 3.4: The Sphinx message structure. The greyed fields are added for the Sphinx protocol.

3. The Source Key: This field stores the key of the source node of the messages. By storing this information we can keep track of the path information, e.g., when a source node requests a signed acknowledgement from a certain host, it queries the node by sending the node its key and sequence number. The node queried has stored this signed acknowledgement in a priority queue under the source key and sequence number.

4. The Sequence Number: Every node increases this sequence number with every message sent.

5. A boolean indicating the message is a secure message: Depending on this boolean, messages are sent secure (following the Sphinx protocol) or unsecure (following the Chimera protocol).

6. The receive time: When a message gets forwarded, this field contains the receive time of the message by the node that forwarded the message.

For every message sent an entry is created in a special sign in queue. This entry contains the time the message was sent, so that action can be undertaken when a signed acknowledgement is not received on time. A field indicating if the message is acknowledged already. The host the message was sent to, when a message is not acknowledged correctly this host can be removed from the routing tables. A copy of the message sent, to send the message to another host when no acknowledgement is received. Every D seconds a special thread checks all entries in the sign in queue for an occurred time out, where D is a system wide configuration parameter. When a time out occurred, the thread is responsible for resending the message.
For every message received an entry is created and stored in a sign out queue. This entry contains the data of the message and a timestamp of when the message was received. Every T seconds a special signing thread signs all messages in this queue, where T is a global system parameter. Before signing a message the node's sign in queue is first checked for acknowledgements for this same message to implement acknowledgement forwarding as described in Section 3.5.3.

Because the signing thread only signs messages every T seconds it is possible to sign more messages at the same time, and in that way increasing the throughput speed and decreasing the network overhead. The drawback of this method is the higher T, the later time outs are detected. The Sphinx algorithm configures T as 1 second.

Chapter 4

Experiments and Results

In this chapter we present the results of our experiments for testing the Sphinx algorithm. In Section 4.1 we describe the experiments we perform and the environment in which they are performed. In Section 4.2 we measure the latency and the maximum time difference between nodes in our testbed. In Section 4.3 we measure the maximum bandwidth in terms of bytes per seconds that can be processed by a node. In Section 4.4 we test the correct operation of a Sphinx DHT under normal conditions. In Section 4.5 we test the detection and locating properties of the Sphinx DHT when malicious behaviour enters the DHT.

4.1 Experimental Setup

The testbed we use as an instance of a corporate WAN is the third-generation Distributed ASCI Supercomputer (DAS-3), which is a wide-area computer system in the Netherlands that is used for research on parallel, distributed, and grid computing. It consists of five clusters of in total 272 dual-processor AMD Opteron compute nodes. The distribution of the nodes over the clusters and their speed is given in Table 4.1. As can be seen, the DAS-3 has a relatively minor level of processor speed heterogeneity. The clusters are connected by both 10 Gb/s Ethernet and 10 Gb/s Myri-10G links for wide-area and for local-area communications, except for the cluster in Delft, which has only 1 Gb/s Ethernet links. On each of the DAS-3 clusters, the Sun Grid Engine (SGE) is used as the local resource manager. SGE has been configured to run applications on the nodes in an exclusive fashion, i.e., in space-shared mode. As the storage facility, NFS is available on each of the clusters.
In order to test the Sphinx algorithm, we will conduct the following four experiments on our testbed:

1. To determine the maximum allowed latency needed for the detection of latency violations, we measure the NTP properties and latency properties of the DAS-3 network.

Table 4.1: Properties of the DAS-3 clusters.

Cluster Location	Nodes	Speed	Interconnect
Vrije University (VU)	85	2.4 GHz	Myri-10G & GbE
U. of Amsterdam (UoA)	41	2.2 GHz	Myri-10G & GbE
Delft University (DU)	68	2.4 GHz	GbE
MultimediaN (MN)	46	2.4 GHz	Myri-10G & GbE
Leiden University (LU)	32	2.6 GHz	Myri-10G & GbE

2. Because the Sphinx algorithm relies on the signing and hashing of messages, which is compute intensive and therefore will influence the bandwidth of a node, we will perform several tests to measure the speed of signing and hashing of messages. We will test the signing and hashing speed of a DAS-3 node, and deduce the maximum bandwidth of a node out of these numbers.

3. To test the Sphinx algorithm under normal conditions where no malicious nodes are in the network, we will set up a Sphinx DHT network consisting of non-malicious nodes, and randomly send messages through the network to test if no node gets the reputation of being malicious.

4. We will test the sensitivity of the Sphinx algorithm by testing how fast a node will be detected and located when it starts to drop, delay, or change the payload of messages. We will use the results of the test where malicious nodes do not return acknowledgements also to prove that the crashing of nodes will not ruin the reputation of its neighbours, because this is essentially the same: crashing nodes also stop returning acknowledgements. In both cases, we need guarantees that the malicious nodes/crashing nodes will not influence the reputation of their neighbours , i.e., no other nodes must get the reputation of being malicious.

4.2 Time Properties of the DAS-3

In this section we derive the time properties of the DAS-3 testbed by measuring the local time differences between nodes, and the round-trip-times (RTT) between the nodes. We need these values to determine the maximum allowed forward latency between nodes.
To get a grip on the expected latency times between nodes, we performed several measurements. We started by measuring the RTTs between the clusters, which is shown in Figure 4.1. Because the network links between the clusters are heterogeneous, we must calibrate the maximum allowed latency on the slowest link, which is the connection between the Delft University cluster and Leiden University cluster.

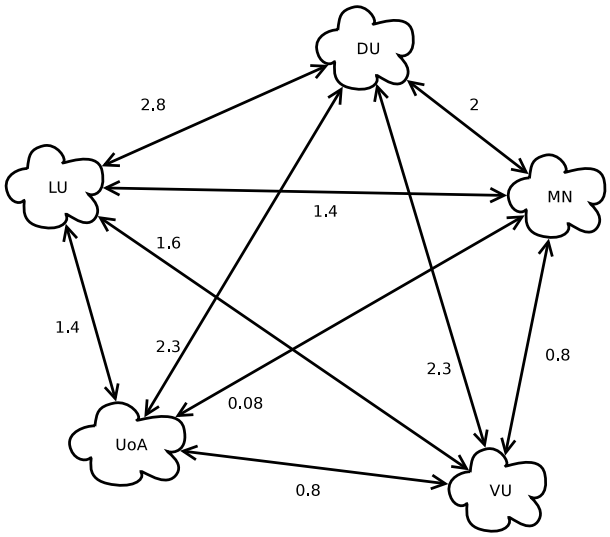

Figure 4.1: Average RTTs in ms between DAS-3 clusters.

Between these two clusters we send 180,000 ICMP-echo packets to measure the RTTs. Figure 4.2(a) shows the cumulative distribution of these RTTs. We also measured the RTTs between the two clusters using an application sending UDP messages. The 99.9th percentile for the ICMP measured RTTs is 3.1 ms, and for the UDP measured RTTs is 5.7 ms. It can be seen that processing data at the application layer influences the latency times, therefore we expect the processing time to be the bottleneck in our testbed.

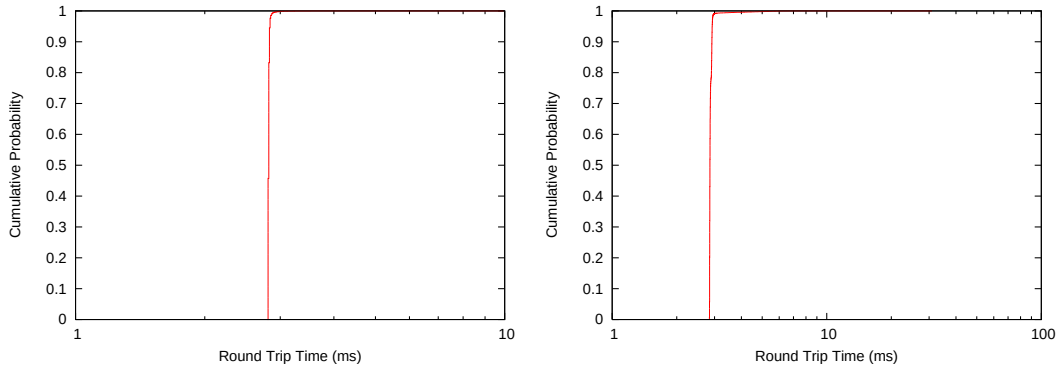

Figure 4.2: CDF of RTTs between two nodes in the DU and LU cluster for (a) ICMP-echo messages, and (b) application layer messages

Every node in the DAS-3 testbed synchronises its time via an NTP server. By using NTP, the time difference between any two nodes in the whole system should

not be more than a few milliseconds [6]. We tested this claim by considering the time differences between one fixed node and 24 random nodes in a DAS-3 cluster. The experiment consists of one node querying the other 24 nodes every second for their local times, by sending them an UDP packet. The nodes return their local times in an UDP packet. By subtracting half the RTT for the messages, from their local times, we get the time differences between the fixed node and the queried node. By plotting these time differences in a graph, we can see the maximum time differences between any two nodes over time..
Figure 4.3 shows the time difference for each of the 24 nodes in relation to the querying node. We can see that the time difference between any two nodes varies between 6 and 2 ms.

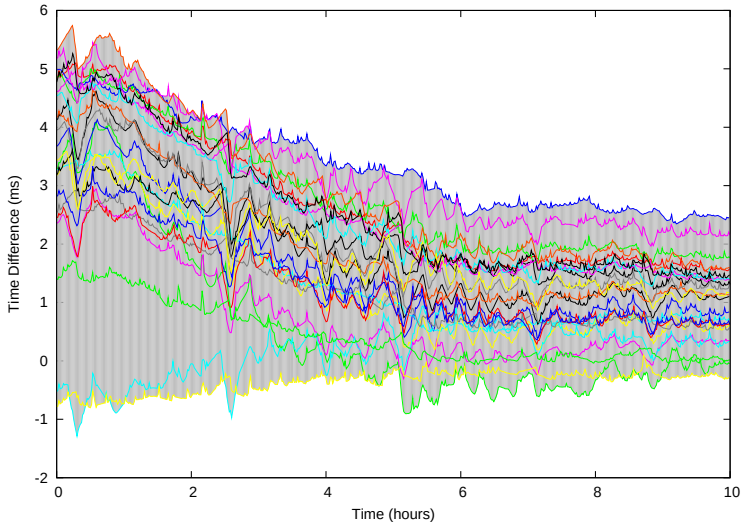

Figure 4.3: Graph shows the time difference between DAS-3 nodes. Every line shows the time difference between one of the 24 random nodes and fixed querying node. The greyed area shows the time difference between the two nodes furthest apart at any time.

Based on these measurements, we set the maximum allowed latency for checking maliciousness to 13 ms, that is, 3 ms forwarding time, 6 ms global time difference and 4 ms processing time.

4.3 Signing and Hashing Speed

The maximum bandwidth of a node depends on the signing and hashing speed, because for every single message forwarded, a node creates a hash. Every acknowledgement sent needs a signature and every acknowledgement received needs to be verified. Hence, the maximum bandwidth of node is defined as the number of messages that can be hashed, signed and verified per second.

We measured the speed of several hashing and signing algorithms performed by a DAS-3 node. The results are shown in Table 4.2. As the hash method, the Sphinx algorithm uses the MD5 algorithm as a good trade-off between security and speed: even though MD5 is proven to be unsecure [15], a hash cannot be forged within the set time boundaries of milliseconds. For the signing algorithm, we choose to use the RSA algorithm, because verification is much faster using the RSA algorithm than using the DSA algorithm, and verification of signatures is more often used than creating signatures, because every node in a message path always needs to verify a received acknowledgement.

As shown in Table 4.2(b), signing is very compute intensive. For instance, using the RSA 512 algorithm with an average packet size of 512 bytes on a DAS-3 node, the bandwidth is $3800 \cdot 512 \approx 1900$KB. Because in the Sphinx algorithm an acknowledgement can cover multiple messages, the bandwidth can be increased. However, this bandwidth cannot be increased linearly, because every single message still needs to be hashed to prove the correct reception of the message to the previous node in the message path.

(a) The bandwidth of different hash algorithms (KB/s)

	Packet Size (bytes)				
Algorithm	16	64	256	1024	8192
MD4	23973	81716	229079	416150	550865
MD5	18498	60908	158363	261065	324501
SHA-1	21248	59442	126422	174665	198391

(b) Sign speed

Algorithm	# Sign/s	# Verify/s
RSA-512	3800.0	46996.5
RSA-1024	1019.5	19228.8
RSA-2048	183.3	6514.3
RSA-4096	29.2	1933.7
DSA-512	6012.8	5245.3
DSA-1024	2243.9	1884.9
DSA-2048	714.8	588.8

Table 4.2: Speed of hashing (a) and signing (b) algorithms performed on a 2.6 Ghz DAS-3 node using the OpenSSL implementation.

Figure 4.4 shows the theoretical bandwidth a node can reach using the MD5 hash method and the RSA signing algorithm with different message sizes, as the number of messages covered by one acknowledgement increases .

The forwarding of signed acknowledgements (Section 3.2) does not increase the maximum throughput in a message path since there is always at least one node that needs to sign the messages and therefore is the bottleneck. However, it does increase the maximum throughput in a node, because the total number of signatures

calculated will be reduced.

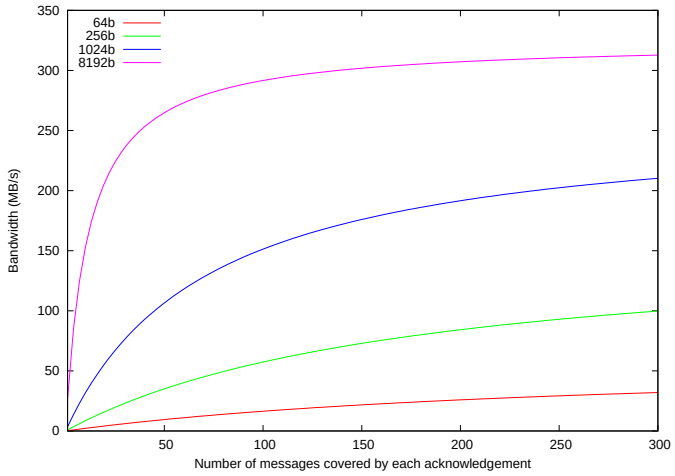

Figure 4.4: The theoretical maximum throughput for a node with different message sizes as a function of the number of messages covered per acknowledgement.

4.4 No Malicious Behaviour

In this section we will test whether normal functioning of the Sphinx DHT will result in nodes getting the reputation of being malicious, which should not occur. Based on our measurements performed in Section 4.2 we set the maximum allowed forwarding time to 13 ms. We set up a DHT network with 28 CPUs on every cluster. On every CPU we run 8 nodes. Hence, in total we create a DHT with 1120 nodes. We run the DHT for 3600 seconds. After a startup period of 300 seconds, each node starts sending a secure 100-byte message to a random destination, every second. Each node measures the number of latency violations for the messages it sent or forwarded. We record the number of messages sent and forwarded, the number of BLAME messages sent, the number of latency violations, and the reputation of every node. To detect malicious behaviour, we use the reply-to-source method, and not the random-check method. The reply-to-source method is more sensitive in detecting malicious behaviour and will suffice in proving the correct functioning of the algorithm. In the test we use only one reputation manager per node and the querying of the messages sent is not done with the "divide and expose" method, but a node will reliably provide the correct number.
It can be seen that in Table 4.3 and Figure 4.5 that 1833 latency violations occurred on a total of $9.5 \cdot 10^6$ messages sent and the lowest reputation did not get below 0.22. Hence, the minimum threshold of 10^{-7} is not reached and therefore no node got a false reputation of being malicious. The number of latency violations is higher than the number of BLAME messages sent, because every node records latency

violations. Also the the source nodes of messages, and a source node will never blame itself. We contribute the fact that the number of latency violations is less than a fraction of 10^{-3} to the uncertainty of the global time differences, processing time, and network heterogeneity.

One hour Sphinx DHT run - 1120 nodes	
Messages sent	3540562
Messages forwarded intra-cluster	1312188
Messages forwarded inter-cluster	4665973
Latency Violations	1833
BLAME messages sent	754
Lowest reputation in DHT	0.022

Table 4.3: The number of messages sent, the number of latency violations, and the number of BLAME messages sent in an one-hour run of a Sphinx DHT with 1120 nodes.

4.5 Malicious Behaviour

In this section we present the results of the experiments conducted with malicious nodes.
We tested four different kinds of malicious behaviour:

1. Dropping messages
2. Delaying the forwarding of messages
3. Not acknowledging messages
4. Changing the payload of messages

4.5.1 Dropping messages

We will test how fast a malicious node will be detected and located when it starts dropping messages, but still returns signed acknowledgements, i.e., it accepts the responsibility for a message.
We set up a Sphinx DHT running on 4 DAS-3 clusters: VU, LU, UoA and DU. In every cluster we use 30 CPU, and on every CPU we run 8 Sphinx nodes. Hence, in total we use 960 nodes. After a startup period of 200 seconds we let every node send a 100-byte message to a random destination. At a predefined time, we let one random node become malicious in the network. From the moment it becomes malicious, the node stops forwarding messages, but still sends signed acknowledgements. The malicious node is not sophisticated; it drops every message it receives. When the reputation of a node drops below 10^{-7}, we consider a node malicious.

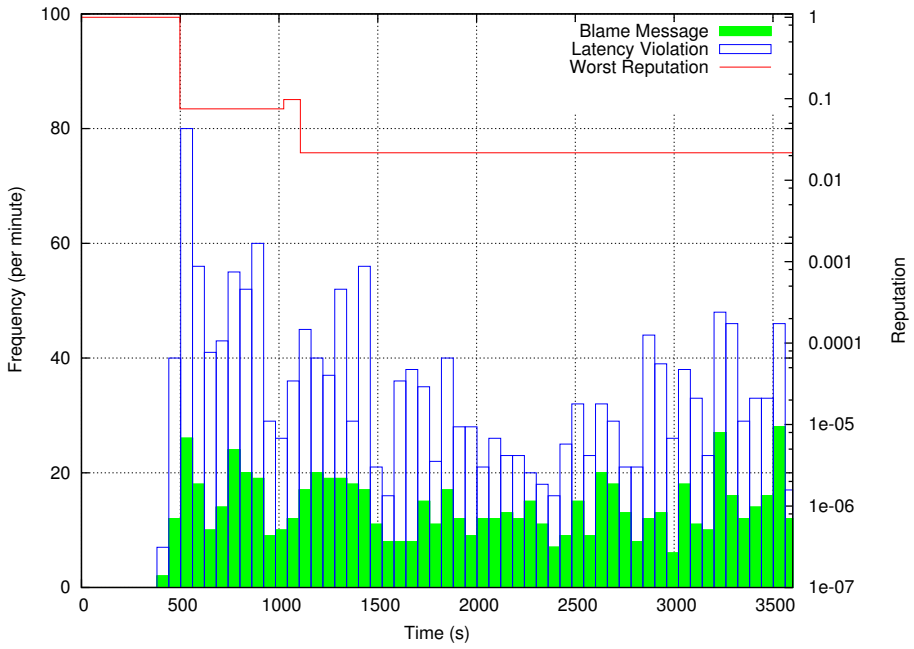

Figure 4.5: Histogram of the number of blame messages and latency violations per minute, and a graph of the reputation of the node with the worst reputation in a normal run of Sphinx with 1120 nodes and a maximum allowed latency of 13ms.

As described in Section 3.4, a node cannot drop more than 4 messages of the last 100 messages it forwarded, before its reputation drops below the threshold of 10^{-7}. It can be seen in Figure 4.6 that as soon as the malicious node start to drop messages, BLAME messages are being sent. Figure 4.6 also shows that the reputation of the malicious node indeed drops below the threshold in 5 steps. The node is now considered malicious. It takes 11 seconds before the malicious node is detected as malicious. In this period of time the malicious node was able to drop 7 messages. The number of latency violations do not increase because the malicious node correctly acknowledges every message.
On average it takes between 4 and 5 seconds for a node to be blamed after dropping a message. This period of time consists of the 2 seconds reply time-out configuration parameter, and the 2 seconds time period within a request for a signed acknowledgement has to be answered. These time-out periods are very accommodating and could be shortened to improve detection times.

4.5.2 Delaying messages

This test shows how fast a malicious node will be detected and located after it starts delaying messages. Again, we set up a Sphinx DHT running on 4 DAS-3 clusters: VU, LU, UoA and DU. In every cluster we use 30 CPUs, and on every CPU we run 8 Sphinx nodes for a total of 960 nodes. After a startup period of 200 seconds,

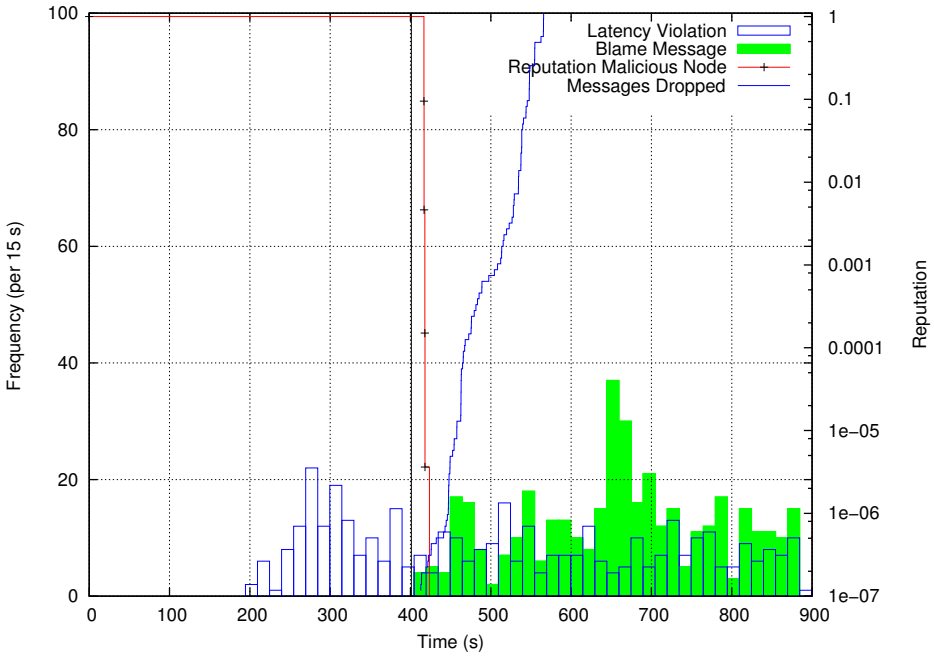

Figure 4.6: Histogram of the number of blame messages and latency violations per 15 seconds, a graph of the reputation of the malicious node plus the number of dropped messages in a run of Sphinx with 960 nodes and a maximum allowed latency of 13ms. The random node becomes malicious at 403 s, drops its first message at 412 s, and has a reputation of being malicious at 423 s.

we let every node send a message each second to a random destination. We let one random node become malicious in the network and let it delay every message, it should forward, for 13ms.

Figure 4.7 shows that within 7 seconds the malicious node is detected, located and marked as being malicious. In this period of time, the malicious node was able to delay 13 messages. It can be seen that blame messages started to be sent when the malicious node starts to delay messages. The latency violations do not increase because the malicious node correctly acknowledges ever message. This test shows that the Sphinx algorithm is able to detect minor fluctuations in forwarding latencies, and to locate the perpetrator.

It should be noted that in this test, the malicious node replies immediately when a source node requests a signed acknowledgement to detect latency deviations. However, the best strategy for a malicious node would be to not provide a signed acknowledgement at all to delay the malicious behaviour detection. This is only makes sense because dropping of messages will not influence its reputation more than delaying of messages. Therefore, in future versions of the Sphinx algorithm the different probabilities for different kinds of behaviour should be taken into account.

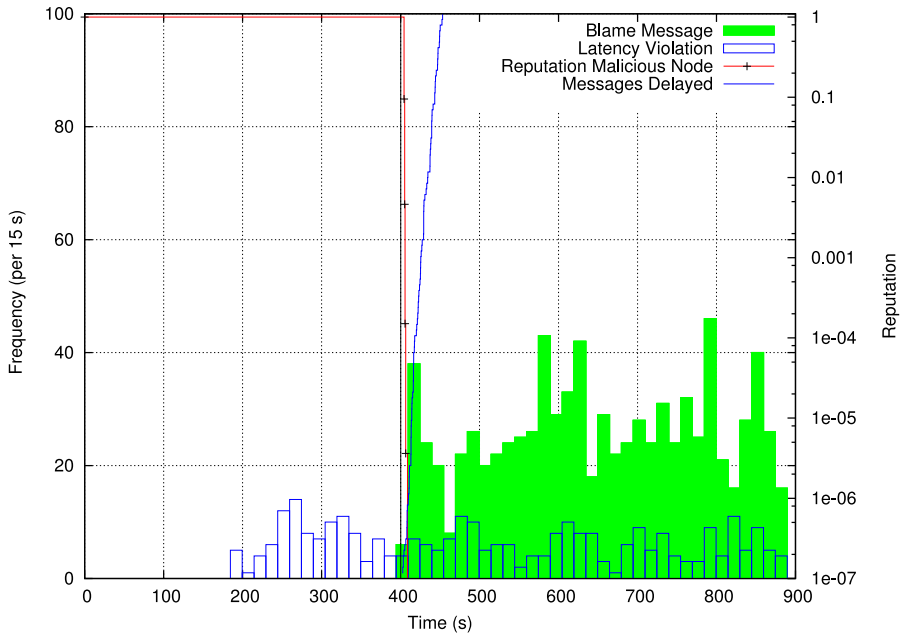

Figure 4.7: Histogram of the number of blame messages and latency violations per 15 seconds, a graph of the reputation of the malicious node and the number of delayed messages in a run of Sphinx with 960 nodes and a maximum allowed latency of 13ms. The random node becomes malicious at 401 s, delays its first message at 402 s, and has a reputation of being malicious at 409 s.

4.5.3 Not acknowledging messages/ Crashing Nodes

When a malicious node stops acknowledging messages, it will influence the reputation of its direct neighbours, because neighbours forwarding a message to the malicious node will need to re-forward the message to a different node. We will set up a test to show that no neighbouring node's reputation will drop below the threshold of 10^{-7}. Since the crashing of a node in the DHT is essentially the same as a node that does not return acknowledgements, this test will also show whether crashing nodes influence the reputation of their neighbours too much or not.

Figure 4.8 shows that the number of messages not acknowledged slowly discrease, this means that the malicious node slowly disappears from the routing tables of all other nodes. Because a node always removes a non cooperating node from its routing tables it will not send another message to the malicious node. The removal out of the routing tables of all the malicious node's neighbours, however, can take a while. Therefore, we recommend as an improvement for future versions of Sphinx, to inform the reputation managers of this misbehaviour. The reputation managers could then check whether the blamed node indeed does not acknowledge the messages it gets. Of course, the blamed node should not be aware if it is a reputation manager that is testing him. Therefore, a reputation manager should not

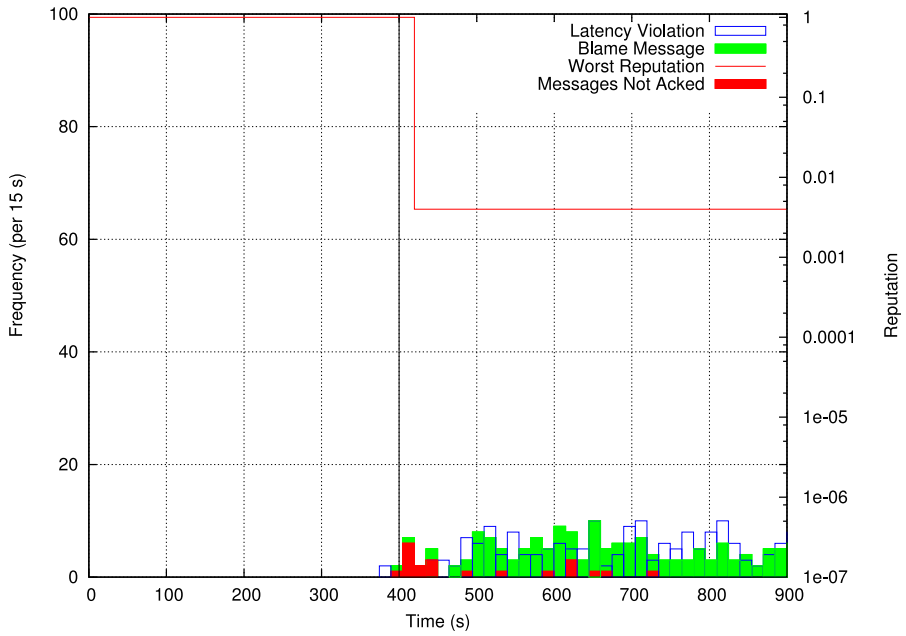

Figure 4.8: Histogram of blame messages, latency violations, and messages not acknowledged, and a graph of the reputation of the neighbouring node with the worst reputation, of a Sphinx DHT run with a malicious node not acknowledging messages.

directly send a message to the blamed node, but through another node.

4.5.4 Changing payload

Because the changing of messages should never happen, we keep to a one-strike-is-out policy. Hence, when the destination replies to the source, the source detects that the MD5 sum of the message does not equal the MD5 hash it calculated itself. The source now starts the locating the node that changed the payload as described in Section 3.3. We will not create an extra graph to show the detection of this malicious behaviour, because the detection and location is essentially the same as detecting forward delays. Only, instead of inspecting the receive times in in the signed acknowledgement, the source node should inspect the MD5 sum in the signed acknowledgements.

4.5.5 Multiple Malicious Nodes

In this section we test how many of the malicious nodes are detected, when we slowly increase the fraction of malicious nodes in the DHT. We set up a DHT containing in total a 1000 nodes. Every node sends a 100-byte message to a random destination each second. Every malicious node in the network returns acknowl-

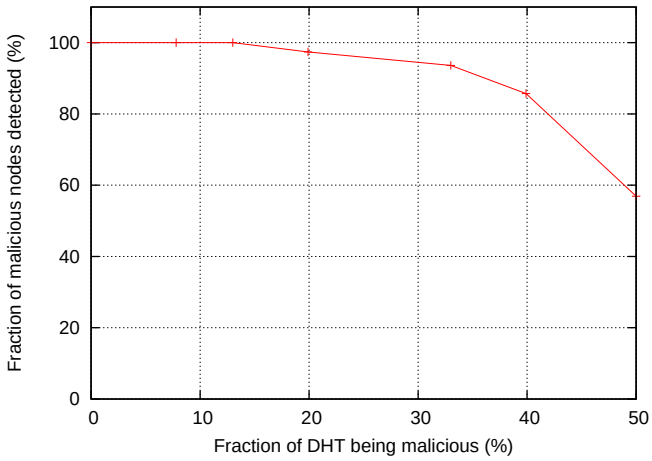

Figure 4.9: Graph of the fraction of malicious node detected as a function of the fraction of malicious nodes in the DHT.

edgements, but drops the messages it receives. By still acknowledging messages, a malicious node will not be removed from its neighbours routing tables directly. A malicious node also ignores the BLAME messages it receives for the nodes it is the reputation manager for. Every nodes has three reputation managers, and a node is detected as malicious if two of its three reputation managers entitle it as malicious. Figure 4.9 shows the results of this test. It can be seen that for a fraction smaller than 10% it is unlikely that a malicious node will not get detected. This complies with our design goals. We did not let malicious reputation managers label non-malicious nodes as malicious, but it can be expected that this number will be the inverse of the graph shown in Figure 4.9.

Chapter 5

Conclusion

In this chapter we will give a summary of the design and implementation of the Sphinx protocol. After this summary we will present the conclusions of this thesis, and we will end with recommendations for future work.

5.1 Summary and Conclusions

In this master thesis we have presented the design and analysis of the Sphinx algorithm, a scalable and robust algorithm capable of detecting and locating malicious nodes in corporate DHTs. We focused on corporate DHTs, because these DHTs often provide business critical services, and because these DHTs usually run in closed corporate WANs. These WANs are company owned and controlled, therefore the network properties are much better known than for general WANs like the Internet. The knowledge of these network properties can be used to detect deviations in routing behaviour of nodes. The Sphinx protocol is able to detect the following malicious behaviour: dropping of messages, delaying of messages, misrouting of messages, and tampering with the payload of messages. Detection and location are both realised within seconds. The heart of the protocol is the use of signed acknowledgements. Nodes that forward messages should receive signed acknowledgements from the next node in the message path. These signed acknowledgements function as 'proofs of innocence'. They enable a node to prove a message is forwarded on time and to the right node. Signing is very compute intensive, therefore the Sphinx protocol implements two ways of reducing the signing of messages: forwarding of acknowledgements, and the acknowledging of multiple messages simultaneously. The Sphinx protocol also provides a scalable reputation system, to keep track of the reputations of the nodes in the network.
From the results of our experiments we can conclude that:

- The use of signed acknowledgements as 'proofs of innocence' is feasible when acknowledgement forwarding and acknowledgement grouping is used.

- The Sphinx algorithm is able to detect malicious delays of milliseconds in forwarding latency.
- The detection and locating of malicious behaviour can be realised with minor overhead and in a relatively short period of time.

This means that it is possible to protect corporate DHTs from intruders by minimising the damage that can be done by detecting malicious nodes in a relatively short period of time, while the correct functioning of the DHT under normal conditions is not disturbed because of the little overhead generated.

5.2 Recommendations

To improve the functioning of the Sphinx DHT, we present the following recommendations:

- The Sphinx algorithm currently only determines whether a node is malicious or not. To make effective use of Sphinx, nodes should query the reputation managers for the reputations of the nodes in their routing tables repeatedly, or reputation managers should notify neighbours of a malicious node. In this way malicious nodes can quickly be removed from the routing tables, and therefore from the DHT.
- The Sphinx algorithm does not yet handle the exchange of certificates that are necessary to verify the signed acknowledgements. Sphinx nodes should exchange certificates with their neighbours, and there should be a protocol for requesting certificates when these are needed.
- When malicious nodes do not return signed acknowledgements, there is no hard proof of the maliciousness of a node. Hence, when this behaviour is detected, the reputation managers of the malicious node should be informed and the reputation managers should start a procedure to determine if the accused node indeed does not return acknowledgements. This of course, should be done in such a way that the malicious node is not aware of being tested.
- The statistics to determine whether a node is malicious or not can be improved. Currently, every latency violation gets assigned the same probability, while larger latencies are of course less likely to occur than smaller latencies. In this way, more accurate estimates can be made about the maliciousness of a node.
- Currently, the Sphinx algorithm does not handle network heterogeneity well: The maximum allowed latency should always be calibrated based on the worst network link. However, one could think of ways where latencies between different subnets can be measured and adjusted in a dynamic way. For

instance, nodes could distribute their average measurement with a gossip protocol through the network. Then, nodes that violate the expected latency times, can be blamed. This dynamic measuring of network link properties may even give us the opportunity to use the Sphinx algorithm in more general WANs like the Internet.

Bibliography

[1] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *International workshop on Designing privacy enhancing technologies*, pages 46–66, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[3] Lakshmi Ganesh and Ben Y. Zhao. Identity theft protection in structured overlays. In *Proc. of 1st Workshop on Secure Network Protocols (NPSec)*, Boston, MA, June 2005.

[4] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[5] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[6] David L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications*, 39:1482–1493, 1991.

[7] Peter Pietzuch, David Eyers, Samuel Kounev, and Brian Shand. Towards a common api for publish/subscribe. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 152–157, New York, NY, USA, 2007. ACM.

[8] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures,*

and protocols for computer communications, pages 161–172, New York, NY, USA, 2001. ACM Press.

[9] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *P2P '01: Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*, page 99, Washington, DC, USA, 2001. IEEE Computer Society.

[10] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[11] Atul Singh, Miguel Castro, Peter Druschel, and Antony Rowstron. Defending against eclipse attacks on overlay networks. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 21, New York, NY, USA, 2004. ACM.

[12] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. Exploiting kad: possible uses and misuses. *SIGCOMM Comput. Commun. Rev.*, 37(5):65–70, 2007.

[13] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[14] Y. Versluis. Secure routing in structured overlays. *Research Assignment TU-Delft*, 2008.

[15] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *EUROCRYPT*. Springer-Verlag, 2005.

[16] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2003.