Survivable Monitoring in Dynamic Networks

Giuseppe Ateniese, Chris Riley, Christian Scheideler
Information Security Institute
and
Computer Science Department
Johns Hopkins University
{ateniese,chrisr,scheideler}@cs.jhu.edu

Abstract

We present a monitoring system for a dynamic network, in which a set of domain nodes shares the responsibility for producing and storing monitoring information about a set of visitors. This information is stored persistently when the set of domain nodes grows and shrinks. Such a system can be used to store traffic or other logs for auditing, or can be used as a subroutine for many applications to allow significant increases in functionality and reliability. The features of our system include authenticating visitors, monitoring their traffic through the domain, and storing this information in a persistent, efficient, and searchable manner. The storage process is $O(\log n)$*-competitive in the number of network messages with respect to an optimal offline algorithm; we show that this is as good as any online algorithm can achieve, and significantly better than many commonly used strategies for distributed load balancing.*

Keywords: monitoring, audit logs, survivable storage, network intrusion detection, emergency communication

1 Introduction

This paper presents a theoretically well-founded, provably efficient monitoring system over a dynamic overlay network. A monitoring system collects and stores information about visiting participants in a network. The information is made available upon request and can be subsequently analyzed and used for any purpose by an administrator, for example to assign or revoke certain privileges for specific monitored nodes which have behaved well or badly.

A monitoring system can be run on a single machine or on several machines (nodes) connected through a network. We assume that such a network is dynamic, i.e., it constantly changes its characteristics. In particular, we focus our attention on unreliable wireless networks, which can be structured (e.g., P2P) or unstructured (e.g., ad-hoc), that support a very dynamic environment where nodes can leave or join and are allowed to move from one point to another.

The set of nodes forming the monitoring system is called *domain* and the nodes inside the domain are referred to as domain nodes. The nodes being monitored are known as *visitors*, and are considered to be outside the domain. We use an adversarial approach which considers an adversary that is able to disconnect any targeted nodes of the domain. However, we assume that the attacker is not able to impersonate any domain node. In particular, we assume that an attacker cannot compromise a trusted node to the extent that secret keys of that node are exposed.

1.1 Assurance Argument

The goal of this paper is to guarantee the collection and availability of monitoring information for activity by visitors in a network. Our solution uses a dynamic set of domain nodes to share the tasks of monitoring the visitors and of storing the generated monitoring information. This information can be used for the detection of a variety of threats, some of which are discussed in the paper. More directly, the technical contributions of the paper consist of preventing disruptions of the monitoring process. Such a disruption could have significant security implications, depending on the application. For example, when using our system to aid network intrusion detection, a disruption of monitoring could allow attackers to cause significant damage in the network before being detected.

The threats to the monitoring process addressed by this paper take three primary forms. First, attacks can cause node failures in the system, which may cause data loss or other disruptions of the storage process. The technique presented in this paper solves this problem by providing a mechanism for efficient data reallocation across domain node joins and leaves, and can also support abrupt node failures through redundancy in connections and data storage. The provable efficiency of the reallocation process also reduces the ability of attackers to overload the system by causing thrashing (repeatedly bringing down and/or inserting domain nodes to cause excessive data movements). Second, excessive information can be generated through malicious overactivity to overload a monitoring node. Our technique solves this by distributing the monitoring information across all participant nodes without overloading any; the information is distributed as closed pages (with no later additions or modifications) so that fixed amounts of storage can be used. Third, nodes can attempt to hide themselves to avoid identification and monitoring. We present one method for preventing such stealth, designed for very low powered devices and small networks such as would be used in an emergency communication system; similar methods are possible in larger networks of devices with greater computational ability using fairly standard cryptographic techniques.

We claim that no monitoring information is lost under the assumption that nodes depart gracefully (with advance notice), or that with high probability no information is lost when nodes are allowed to depart without notice; in addition to this we claim that no node is ever asked to store more information than it is capable of storing (or has stated that it is capable of storing). We also claim that no traffic is allowed to travel through the domain nodes that is unmonitored. We assume that we have a reliable and trusted supervisor node to manage the organization of the network of domain nodes, though when a multicast group of spare supervisors is kept only one needs to remain to keep the system intact. We also assume that domain nodes are trusted (though they do not need to be reliable).

Attempts to tamper with the system or bypass the monitoring activity are met with cryptographic solutions. Methods for authentication ensure that visitor nodes are identified before being allowed to communicate; message encryption within the network ensures that no node can impersonate a domain node or send messages through the domain nodes except through the proper monitoring process. While there are many valid solutions to these problems, a sample solution based on low-powered small networks (such as ad-hoc networks of cellular phones and PDA's) is presented.

1.2 Applications

Depending on how the information collected by the monitoring system is used, there are several applications that could benefit from our strategy. In this section we consider three such applications: persistent audit logs, network intrusion detection, and emergency systems.

Persistent Audit Logs. Audit logs of activity in networks can be considered a type of monitoring information, so our system can be used with no modifications to address this concern. The generic monitoring process involves using a domain node to intercept visitor traffic and process it to acquire the application-defined monitoring information; this can be used to perform any desired auditing. Our system produces audit information locally and then stores it persistently and efficiently in a highly dynamic set of nodes.

Network Intrusion Detection. The primary goal of a network intrusion detection system is to collect information on entities specified by a system administrator or by a policy and to analze this information against a set of filters. Each node in the network contributes to the collection effort. There are two types of network detection systems, depending on where the analysis occurs. In the first type of system, nodes collect the information and send it to a central database where the information is analyzed. In the second type of system, each node analyzes the information locally and notifies all nodes if a problem is found.

Our system could be used as an information collector for a system of the first type. Network intrusion detection systems focus on protocol flow analysis with the purpose of determining whether some entity has attempted to gain, or has gained, unauthorized access to the system; our system would provide the intrusion detector with the needed information to perform its analysis, and would ensure that the information was not lost. Our model allows adversarial disabling of nodes in the network without losing information. Therefore, our monitoring system combined with network intrusion detection techniques can be used as a *survivable* intrusion detector that can withstand network failures and malicious node disabling.

Emergency System. In emergency situations, such as transportation strikes, terrorism attacks, and natural disasters, an emergency response information system is employed to coordinate the activities of several individuals from different organizations. Emergency systems are designed as structured group communication systems with an integrated electronic library of external data and information sources [37]. Future scenarios may force such systems to extend their requirements to consider new threats that were not considered before. For example, terrorist organizations could create an emergency situation and at the same time destroy any communication systems used by rescue or reporting teams, consequently amplifying the terror effect. A communication infrastructure is a relatively easy target and can be disrupted without much effort. Thus, emergency systems should not rely exclusively on communication networks containing critical components with geographically fixed positions. Even distributing a component by splitting its role among several nodes spread around the globe may not be sufficient, since it is possible to isolate an entire geographic area where the emergency is taking place. Even if the rest of the communication network works properly outside this area thanks to the distribution, local emergency operators (such as rescue teams) will not be able to communicate within the area. Next generation emergency systems must be designed to support a dynamic set of members that includes low powered and sporadically connected participants such as cellular phones and PDAs; systems must be able to use whatever resources are available and must be able to tolerate their weaknesses.

Current emergency systems are *closed* by design, i.e., only a predetermined set of experts are allowed to exchange information [37]. However, it is impossible to predict how a crisis situation might evolve. It is impossible to completely predetermine the roles and responsibilities of participants in the network or even the set of individuals or organizations allowed to use the system. An open approach would provide several benefits by allowing new members from different organizations to use the communication infrastructure provided by the emergency system and contribute to the crisis management process. However, *external* members (i.e., members that are allowed to use the system only because of the emergency situation) should at least initially be restricted or monitored to avoid any abuse of the network.

Our monitoring system allows untrusted nodes to communicate while enforcing that their behavior is monitored by the domain nodes. Furthermore, our system is tolerant of changes to the set of trusted participants as well as to a wide range of capacities. This makes it particularly suitable for serving as the monitoring process for future emergency communications systems running on ad-hoc networks that allow open but monitored communication.

1.3 Related Work

Emergency communication systems are becoming increasingly popular but, to the best of our knowledge, none of the existing and proposed systems operates over a dynamic network and provides an open access policy that allows visitors to communicate; see [37] for a survey of existing emergency systems.

There has been extensive research in intrustion detection for at least the past twenty years, see for instance [1, 4, 7, 14, 15, 16, 17, 19, 21, 22, 32, 38]. In particular, intrusion detection for distributed systems is a very active research area and several systems have been proposed that can be classified based on the approach employed by the detector. For instance, DIDS [33] and NSTAT [18] are systems based on the centralized analysis approach where audit data is collected on individual nodes and then reported to a centralized location where the intrusion detection analysis is performed. In GrIDS [34] and EMERALD [26], systems based on the hierarchical analysis approach, audit data is collected and analyzed by each node and the results of the analysis is reported according to some hierarchical structure.

The technical contribution of our paper is represented by the load balancing and recovery mechanisms built on top of the overlay network SPON [29], which was designed for reliable broadcasting in dynamic networks. Extensive research has been recently carried out on the design of overlay networks that support arrivals and departures of nodes. Recent systems projects on such networks include Freenet [9], Ohaha [24], Archival Intermemory [8], and the Globe system [3]. Theoretically well-founded peer-to-peer networks have also been presented, such as Pastry [30], Tapestry [20], Chord [36], and a network presented by Pandurangan et al. [25], along with SPON. With the exception of SPON, the topologies of these networks are based on either DNS-like, hypercubic, or random constructions, which are either not useful or far too complex for our particular environment. Recently, a new backup system based on peer-to-peer overlay networks has been proposed in [10], similar to an approach previously suggested in other works, including, for example, [5, 11, 12, 28, 31]; the scope of these systems is to backup entire file systems. The storage component of our system, designed solely to store monitoring information, allows us to fulfill our requirements while achieving provable efficiency, which more expensive systems cannot.

1.4 Problem description

We assume that there are two different kinds of nodes, *visitors* and *domain nodes*, and that the visitors are untrusted and the domain nodes are trusted. The task of the domain nodes is to monitor all activities of the visitors which involve the network. They also store a distributed database containing recorded monitoring information for all visitors.

There are three components to this monitoring process:

1. All traffic of the visitor has to be intercepted.

2. The intercepted traffic must be processed to produce relevant monitoring information.

3. This information must be stored permanently.

This paper focuses primarily on the last of these, proposing a distributed database and algorithms for the storage of this information. The requirements of such a database are as follows:

1. **Authentication:** The system must be able to identify visitors accurately to ensure that stored information can be correctly matched to a visitor.
2. **Searchability:** The database must be searchable, in the sense that an administrator must be able to acquire all information about a particular visitor wishing to connect to the network.
3. **Persistence:** The database must be persistent, in the sense that no entries in the database can be lost by network disruptions.
4. **Efficiency:** The algorithms for maintaining and using the database should run with minimal communication and computational overhead.

1.5 Model

We assume that we have available a dynamic set of domain nodes which can be used to store the database. These nodes may depart the network at any time, but for most of this paper we assume that node departures are *graceful*, or that each node requests permission to depart and does so only when told. This allows the analysis to be less dependent on a sensitive failure model for nodes. It is not a limiting assumption, as the algorithms presented here can be extended through redundancy to limit the probability of node failure through ungraceful departures. See Section 5 for details.

For clarity we will not concern ourselves with network layer details. We assume that all domain nodes participate in some network which supports broadcast and unicast message passing; if the underlying network supports only unicasting, a separate SPON can be used for broadcasting. We do not consider latency across edges, edge bandwidth, or edge failures. The performance metric we are concerned with is the number of bits transferred through the network in the monitoring system, which includes control information and any data movements used to rebalance the system or to move data to a new node or from a departing node.

We assume that each domain node has a limit on its capacity for storage, to make the storage problem nontrivial; for analysis purposes we will assume that these capacities are identical for all nodes, but the system functions without modification with varying capacities. We assume that the sum of the capacities of the domain nodes is more than the amount of information being stored in the database, so that the system never runs out of space.

We assume an adversarial event model, where an adversary is allowed to request node join and leave events. This adversary can be used, for example, to always request that the most heavily loaded node leave, causing all the data stored at the node to be moved elsewhere. Within this model we will prove competitive ratios comparing our algorithm to an optimal offline algorithm which knows the entire sequence of events. More details on this analysis method can be found in Section 4.1.

We assume that there is a single supervisor node, which can be of low bandwidth, storage, and processing power. To ensure reliability, we also assume that a constant-sized multicast group of spare supervisor nodes is maintained. When communications are sent to the supervisor we assume they are sent to the multicast group address. All nodes in the group except the active supervisor ignore traffic in the group not sent by the active supervisor. The active supervisor multicasts updates in its stored information to the group to keep the spares up to date; then if the active supervisor fails, any one of the spares can take over transparently. Central organization is integral to the performance of our algorithms; there are no known symmetric techniques which satisfy all our requirements.

1.6 New results

We present a design for a system which is capable of authenticating visitor nodes, monitoring their traffic through a network of domain nodes, and storing this monitoring information in a persistent and efficient manner. The system supports searching for data collection through network broadcast. Our system can be used as a component of an emergency communications system or an intrusion detection system or many other systems, depending on the analysis performed on the collected information.

Our system provides an online algorithm for data storage which is $O(\log n)$-competitive against an optimal offline algorithm[1], or that any algorithm which knows the future sequence of visitor communications and node departures and joins is at most a factor of $O(\log n)$ more efficient in network communication, where n is the number of domain nodes (or the maximum number in a relevant time interval). Furthermore, we show that this competitive ratio is asymptotically as good as any online algorithm (one which does not know the future), and is notably better than many commonly used strategies.

[1] See Section 4.1 for a more thorough description of competitive analysis

2 System Overview

2.1 Motivation

Clearly the nearest domain node to the visitor at the time of communication must be the one to intercept the visitor's messages, since if the messages passed the first domain node without being caught they would be considered to have entered the network without being monitored. Beyond this, there are a number of simple solutions to the problems described above which fail to achieve the desired objectives; their failings motivate our solution. As one example, after being produced, all monitoring information for all visitors could be immediately sent to a single domain node to be stored. This is unsatisfactory because it would almost certainly overload the node. Obviously, it is better to divide the data storage among all domain nodes. One method would be to select a single domain node for each visitor, for example through distributed hashing[36], and to send all monitoring information to that node. But some visitors may generate much more monitoring information than others, and some domain nodes may have multiple high traffic visitors assigned to them, potentially exceeding the capacity of the node.

Another solution is to let the domain node nearest the visitor store the monitoring information for the visitor. But the visitor may move to other places in the network, and the nearest domain node may change frequently. The monitoring information could be exchanged, but this would generate significant communication overhead. The monitoring information could be left at the domain node that collected it, and collected only when needed; this saves unnecessary message passing, but can cause load imbalances and can exceed the capacity of domain nodes.

The system presented in the following sections does not have any of the problems of these strategies.

2.2 Components

This section discusses the visitors, the domain nodes, and the supervisor, mentioning their responsibilities and providing sample interfaces.

2.2.1 Visitors

The visitor is responsible for authenticating its messages by signing them, so that a domain node receiving them can properly match the traffic to the node. Any unsigned messages from a visitor are ignored.

VISITOR INTERFACE
sign(): authenticate itself in its messages

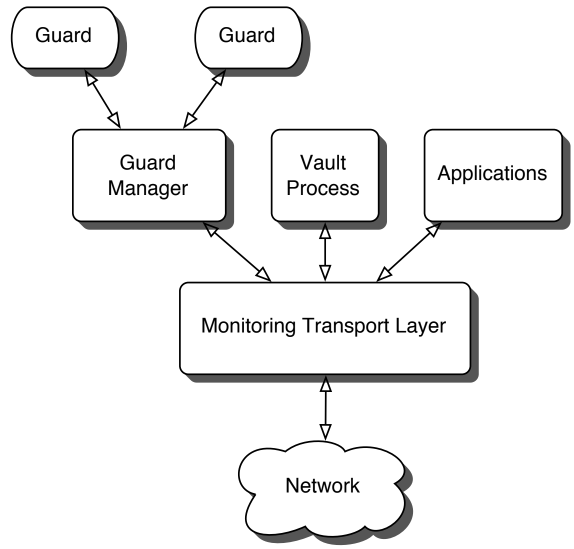

Figure 1. Layers of the Secure Monitoring Protocol

2.2.2 Domain nodes

An overview of the protocol for an ordinary domain node is given in Figure 1.

The monitoring transport layer receives all messages arriving from the network. It passes messages which are not valid domain messages to the guard process, and routes valid domain messages to the guard and vault processes and to any applications in use (and to the supervisor process in the supervisor node) according to their destination. It also signs all messages from the node, from any process, to mark them as valid domain messages.

MONITOR TRANSPORT LAYER INTERFACE
sign(): sign all outgoing messages as valid domain messages *route*(): route incoming messages to appropriate processes *˜Monitor*: destructor process, calls other destructors

The guard process verifies the identity of a visitor and clears it with the supervisor when it first connects. On the first and subsequent connections, the guard forwards the visitor's messages into the network, and also produces monitoring information about the visitor's messages. We use a single *guard manager* in the domain node which spawns independent guard processes for each visitor connecting through it.

GUARD MANAGER INTERFACE

new(): spawn a new guard process to handle a new visitor
delete(): delete a guard process
˜Guard-Manager: destructor process, delete all guards

GUARD INTERFACE

check(): query the supervisor regarding a visitor
monitor(): produce monitoring information
forward(): send a visitor's message through the network
page(): request a node to send monitoring information to
upload(): send monitoring information to a vault
˜Guard: destructor process, send partial information to a vault

The vault process is responsible for the storage of monitoring information assigned to it. It also participates in an overlay network organized into a distributed heap for the purpose of allocating monitoring information to vaults in a balanced way.

VAULT INTERFACE

join(): join a heap
leave(): leave a heap, for example when full
page(): request a node to send monitoring information to
move(): move data to another vault
heapify(): rearrange with neighbors in heap
search(): search locally stored information for a specific node
write(): write monitoring information locally
˜Vault: destructor; call leave(), move data away with page() and move()

2.2.3 Supervisor

The supervisor is a single domain node known by all other domain nodes, and is also a process running on that node which performs the supervisor functions. The supervisor process serves as the maintainer of the heap of vault nodes. It also stores a blacklist of visitors which are not allowed to send messages through the domain, though it is not responsible for determining the list; traffic analysis is beyond the scope of this paper.

For fault-tolerance, we use a constant sized multicast group as the network supervisor. A single node in the group serves as the active supervisor, and the other nodes ignore multicast messages from outside the group. When the active supervisor changes its stored information about the network, it multicasts the updates to the other nodes in the group; if the group is constant-sized this is at most a constant multiple overhead. If the active supervisor should fail any node in the group is capable of taking over transparently and with no communication overhead.

SUPERVISOR INTERFACE

insert(): add a vault to the backup heap
remove(): remove a vault from its heap
check(): see if a visitor is in the blacklist
update-list(): update the blacklist when told
get-lightest(): return the top vault in the active heap
switch-heap(): activate the backup heap

2.2.4 Administrator

The exact functioning of the administrator is beyond the scope of this paper. In general, the administrator initiates data collection through broadcasts through the domain, in order to retrieve all monitoring information about a set of visitors. If broadcasting is not a primitive in the domain, a strategy such as [29] can be used to perform reliable broadcasting using a unicast primitive.

2.3 Flow path of messages

Messages can be freely exchanged between domain nodes. A message from a visitor node is stopped at the first domain node it reaches (which may change over time as the visitor and domain nodes move around), and the node determines whether or not to let the visitor send to the network by contacting the network supervisor. The domain node monitors the traffic of the visitor after it is cleared by the supervisor. Monitoring information is distributed through the domain by being sent piecemeal through the network to vaults, and can be accessed and used by an administrator, for example to update the network's acceptance policies for visitors.

A sample overview of the flow of a message is given in Figure 2.

3 Algorithms

3.1 Cryptographic Algorithms

Effective monitoring is only possible if untrusted nodes cannot create multiple or false identities, and if the com-

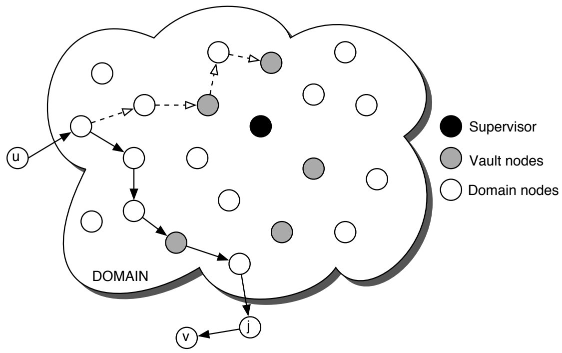

Figure 2. The flow of a message from a visitor through the network to another visitor. The solid path is the message path, and the dotted path is the path of some monitoring information.

plete traffic to and from an untrusted node is monitored, filtered, and stored.

Central requirements for a monitoring system are:

1. **Untrusted nodes must be uniquely identifiable.** This can be achieved via a wide range of standard authentication techniques, from password-based systems to digital certificates that bind node identifiers to public keys. (This is similar to the unique network identifier in intrusion detection systems [33, 18].)

2. **Domain nodes should be able to communicate securely.** Domain nodes should be able to communicate so that outsiders cannot read, modify or inject messages. This can be achieved via standard techniques although techniques based on public-key cryptography should be kept at minimum whenever domain nodes are mobile, since mobile nodes often rely on battery power which can be consumed rapidly by CPU-intensive operations.

Depending on network conditions there are a number of standard solutions to these requirements, including public-key cryptography, shared keys, and group key communication protocols. In Section 6 we discuss a set of solutions to these issues, designed for a single application. This section by no means represents the only way to implement the general system presented in this paper.

3.2 Data Management Algorithms

3.2.1 Guards, pages of logs, and temporary page storage

As the guard monitors the visitor, it stores this information in a temporary fixed size page of storage space; when this page is filled, the guard requests a destination from the supervisor through *page()*, receives a network address, and calls *upload()* to send the page to the address to be stored in that node's vault process. The guard's temporary page can then be erased and reused. Collecting the data into pages improves the efficiency of the supervisor, since each store operation includes a certain overhead cost independent of the amount of data being stored. But if a page is too large, or if all data is stored at the guard, then the load can become unbalanced.

3.2.2 Vaults and SPON-based heaps

For permanent storage of pages, vaults are organized into heaps based on the SPON network presented in [29]. SPON stands for "Supervised Peer Overlay Network", and is a topology and algorithms to maintain a broadcast group in a dynamic network where nodes join and leave frequently. The SPON topology includes a single reliable supervisor, which is responsible for network repair and for initiating broadcast messages. The supervisor stores $2 \log n$ *root slots* organized into $\log n$ pairs labelled from 0 to $(\log n - 1)$. These slots each can either hold a node or be empty; a node in a root slot is a *root node*, and other nodes are *tree nodes*. If a slot in pair i holds a node, then this node is the root of a distributed full binary tree of depth i; it maintains a connection to two other nodes, each of which maintains a connection to two other nodes, and so forth. This is an invariant in the structure of the network, such that after join and leave operations are fully processed this condition still holds. The relevant properties of SPON are that each node has a constant maximum degree[2], the network has $O(\log n)$ diameter, and node join and graceful leave operations can be fully processed in $O(1)$ rounds with $O(1)$ messages (with very low constants).

In this paper we consider right siblings of root nodes in the SPON topology to be parents, and left siblings of root nodes to be children, so that nodes are only connected to a single parent and have up to three children, and all nodes but the rightmost root node have a parent; this does not change the functionality of the network in any way, but allows us to treat the entire graph as a single tree.

[2]The supervisor stores the slots and the nodes within them, but only needs to maintain an edge to the end nodes, though it communicates in response to a direct contact with other nodes.

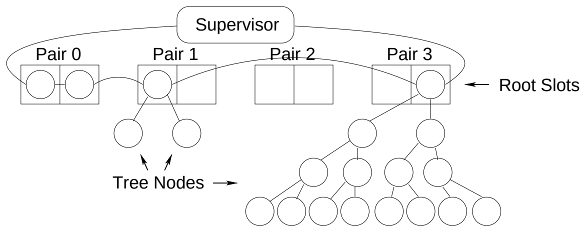

Figure 3. A sample SPON

On top of this topology a heap is maintained, where heaping is performed according to the maximum space available at a node, such that each node has at least as much free space for storage as its children. This is done through the *heapify()* calls of each node, which need to occur only when a node joins or leaves (when its replacement is inserted) or when a node is given additional load. An inserted node queries its new parent and children (as applicable). If it has more free space than its parent, they exchange places by exchanging adjacent node information and informing their neighbors as well; this requires $O(1)$ rounds and messages. Then the node continues to move itself up the tree querying its new parent and exchanging until a terminal location is found; each round requires $O(1)$ messages and rounds of communication, and the supervisor does not need to be involved in any of the operations. If an inserted node or a node given additional load has less free space than its children, it exchanges places with the child of most free load, and continues to query its new children and exchange until it is in place. In this way the root of the rightmost tree is always the node with the most room.

3.2.3 Types of vaults and heaps

There are three types of vaults: *old*, *spare*, and *active*. There are also two separate heaps maintained in the system, the *active* heap and the *backup* heap. Active vaults are connected to the active heap, and spare vaults are connected to the backup heap; old vaults are not in a heap. The root node of the active heap is sent by the supervisor to the next guard node to request storage for a page.

3.3 Event processing

There are several events which may occur. The events and the processes for handling them are as follows:

1. **First contact:** *A visitor contacts a domain node which is not currently monitoring it and wishes to connect to the network.*
   The guard monitor running on the domain node spawns a new guard process for the visitor. If too many guards are running, the guard manager can delete the least recently accessed guard to make room. The guard process checks with the administrator to see if the node is allowed to participate. If okay, then the guard creates a temporary page for the node and begins monitoring it.

2. **Stop contact:** *A visitor currently being monitored by a guard node moves to another guard node or leaves entirely.*
   The guard does not need to perform any action to respond to this. It may not even be able to detect such movement.

3. **Guard's temp page full:** *The page being used by a guard to monitor a node fills.*
   Request a domain node address from the supervisor, send the full page to the node and erase the temporary page for reuse. The vault receiving the page reheaps itself.

4. **Supervisor receives a request for storage:** *A guard asks for a node to store a page.*
   The supervisor replies with the address of the node at the top of the active heap.

5. **Vault receives a page to store:** *A vault receives a page of monitoring information to store.*
   Store the page and then reheap with children until the heap property is restored.

6. **Guard departs:** *A guard process wishes to leave the network.*
   The pages in the node must be placed elsewhere in the network before the node can depart. It requests nodes from the supervisor for each page.

7. **Old vault departs:** *An old vault (not in the active or the backup SPON) wishes to leave the network.*
   This is identical to a guard departing the network.

8. **Spare vault departs:** *A vault in the backup heap wishes to leave the network.*
   The vault has no data to be moved, but it is a part of a heap, so the SPON leave procedure must be used to repair the overlay network.

9. **Active vault departs:** *A vault in the active heap wishes to leave the network.*
   First, the node must be extracted from its heap like a spare vault. Its replacement must reheap itself either up or down as appropriate. Second, its data must be replaced into the system like an old vault and guard.

10. **Node joins:** *A new vault joins the network.*
The node is integrated into the backup heap through SPON's join procedure, and if needed reheaps itself.

11. **Active vault fills:** *An active vault fills its storage space (has less than a page free).*
Nothing different is done; the node is rotated down the heap as in ordinary operation.

12. **Active heap full:** *The supervisor cannot place a page into the active heap because all nodes are full.*
The active heap is dissolved, through a broadcast message or implicitly, and all active vault nodes become old vault nodes. The backup heap becomes the active heap, and a new, empty, backup heap is created. Any data which could not be stored in the previous heap is stored in the new heap, which by assumption must have room for the data.

4 Analysis

4.1 Tools

We use *competitive analysis* as our measurement of the efficiency of our algorithm. Competitive analysis is a standard measure in theoretical computer science for evaluating online algorithms. It compares the work performed by the given algorithm to the minimum amount of work that an optimal offline algorithm must perform in the same setting, where the offline algorithm receives the entire sequence and timings of operations in advance. By proving an upper bound on the ratio between these costs, one can show that there are no "bad" cases where the algorithm performs arbitrarily badly.

Formally, for any instance (sequence of operations) I, for an optimal offline algorithm OPT, for a $cost_I()$ function measuring the work performed by an algorithm to process instance I, and for an appropriate parameter n, to say that algorithm A is $O(f(n))$-competitive is to say that:

$$\frac{cost_I(A)}{cost_I(OPT)} = O(f(n))$$

In addition, *constant-competitive* says that $O(f(n)) = O(1)$.

In this paper, the instance I represents a sequence of node join and leaves and data write requests interspersed. The $cost()$ function counts the number of distinct messages which must be sent through the network to process the instance; it does not attempt to count the number of hops each message takes or the edge latency or total time of message transfer, as these are beyond the scope of this paper. The parameter n is the maximum number of domain nodes in the network at any given time.

The cost in network traffic of our algorithm can be broken into two types, *control messages* and *data movements*. Control messages include, for example, a guard requesting a node from the supervisor to store a page, as well as the supervisor's response. Data movements occur when a page of data is exchanged between nodes, either from a guard to a vault or from a vault to another vault before departing.

4.2 Control messages

We analyze the cost of control messages through the following comparison to the cost of data movements.

Lemma 4.1 *Except for messages to process nodes joining and leaving heaps, control message cost is at most a constant multiple of data movement cost.*

Proof. Other than nodes joining and leaving heaps, control messages are triggered by two types of events, visitor communication and page movement. The initial communication of a visitor to a guard causes the guard to check with the supervisor to see if the node is okay; this is a constant overhead. But an optimal algorithm still must send the messages from the visitor to a node, and since each message can only generate a constant overhead, the algorithm is constant competitive.

Page movement control messages are identical regardless of whether a page is moved from a guard to a vault or from one vault to another. The source process must request a destination node from the supervisor, which responds, and then after moving the data the destination node may need to heapify itself. This total process requires up to $O(\log n)$ messages. But a page of data is moved in the process, so if we assume a page is of size $\Omega(\log n)$, the total cost of control messages in page movement is a constant multiple of the cost of data movement. Since page size is arbitrary the lemma holds. □

In addition to this lemma, node join and graceful leave operations can be processed with $O(1)$ messages in SPON.

4.3 Data movements

We will compare our algorithm to an optimal offline algorithm OPT which knows the entire sequence of operations. We consider our vaults to have an additional page of storage beyond the vaults in OPT; this prevents bin packing problems from causing competitive ratios to be unbounded.

When data is written to the active heap in our algorithm, the optimal algorithm OPT may instead write the data to a different node in the active heap or to a node in the backup heap. Let us consider a suboptimal extension of OPT, SUB, which always writes the data first to a node in the active heap; this is always possible since the active heap is by definition not full, since if it fills it stops being active. If OPT would have assigned that data to a node currently in the backup heap, then SUB moves the data to that node when its first node fails. Clearly the cost of SUB is at most twice the cost of OPT under any circumstances.

Lemma 4.2 *The amount of data in the vaults in the backup and active heaps in our algorithm is at most the amount in the same vaults in SUB.*

Proof. This holds because in our algorithm all nodes not in the backup and active heaps are full (in the sense of having less than a page free), and consequently are holding at least as much information as in SUB. This holds for OPT as well. □

Lemma 4.3 *Data movements caused by departures of vaults not in the active heap is constant competitive to SUB.*

Proof. At time t, let A be the set of vaults in the active heap, B the vaults in the backup heap, S the set of all old vaults (not in either heap), and V the entire set of vaults, so that $V = A \cup B \cup S$. Of the load stored in $A \cup B$ in both algorithms, some will have been first placed in $A \cup B$ and some will have been moved in when a vault in S departed. Because SUB places data first in the currently active set, the amount of load in $A \cup B$ placed in $A \cup B$ to begin with is the amount of load placed in A to begin with, which is the same in both algorithms since both first place all load in A. According to Lemma 4.2, SUB must have at least as much load in $A \cup B$ as our algorithm. Therefore SUB must have moved at least as much data into the active heap from vaults not in the active heap as our algorithm. □

Lemma 4.4 *Data movements produced by the departure of a vault in the active heap are $O(\log n)$-competitive to SUB.*

Proof. Consider the maximum load L^* being stored in any active heap at any point in time while it is active. The active heap begins with n_0 nodes and ends with n_f nodes when the system fills, where $n_0 \geq n_f$, since it can only lose nodes; also, since $n_0 - n_f$ nodes failed while the system was active, $n_0 - n_f$ movements of data within the active heap occurred.

The properties of the heap imply that if there are k pages being stored in n nodes in the active heap, every node will have at least $\lfloor \frac{k}{n} \rfloor$ and at most $\lceil \frac{k}{n} \rceil$ pages. We assume that the load is sufficient to avoid severe discretization effects; in particular, $k \geq n$. Then if there are n nodes currently in the active heap storing up to L^* load, the cost of a node failure is at most $\lceil \frac{L^*}{n} \rceil \leq 2 * \frac{L^*}{n}$. The total cost of all data movements within the active heap is at most:

$$\sum_{i=n_0}^{n_f} 2\frac{L^*}{i} = 2L^* \cdot (\log n_0 - \log n_f) = O(L^* \cdot \log n)$$

According to Lemma 4.2, SUB must also store at least L^* load in $A \cup B$, which must first have gone through A. Then SUB required messages proportional to L^* either to move the data into A or to transfer the original visitor messages generating the load. □

The above lemmas allow us to show the following theorem:

Theorem 4.5 *The load balancing algorithms are $O(\log n)$-competitive to an optimal offline algorithm OPT.*

Proof. Lemmas 4.2, 4.3, and 4.4 show that the algorithms are $O(\log n)$-competitive to SUB with respect to data movements. Lemma 4.1 shows that the cost of most control messages is a constant multiple of the cost of data movements, so the algorithms are $O(\log n)$-competitive to SUB with respect to all messages. Control messages used to support vaults joining and leaving heaps require only $O(1)$ messages. When a vault leaves a heap because it is full, the messages can be absorbed within the data movement cost needed to fill the vault. When a vault leaves a heap because its parent node leaves, any algorithm needs to support the node leaving the network, and would need to inform some other node with at least a message. Since SUB is 2-competitive to OPT, the theorem holds. □

We also have the following lower bound:

Theorem 4.6 *For any online algorithm it is possible to construct a situation for which the algorithm is at best $\Omega(\log n)$-competitive.*

Proof. We give an example for which an optimal offline algorithm can store a load of amount L with work $O(L)$ but any online algorithm must invest effort $\Omega(L \cdot \log n)$. Initially n nodes are available to store a load of amount L, but only any k for some constant k are needed to store the load. An adversary for any online algorithm would sequentially delete $n - k$ nodes by deleting a node of maximum load at each step (causing that node's load to be moved to other nodes). The minimum cost of an online algorithm comes from distributing the load evenly at every step, so that when there are i disks still in the system at least $\frac{L}{i}$ needs to be performed at the next step. In sum:

$$\sum_{i=n}^{k} \frac{L}{i} = L \cdot (\log n - \log k) = \Omega(L \cdot \log n)$$

Since an offline algorithm can store the load initially at the last surviving k nodes for any sequence of deletions, it only exerts L effort, and the competitive ratio is $\Omega(\log n)$. □

Techniques such as distributed hash tables [36] are even worse. DHT's and many similar strategies attempt to include a node in the load balance as soon as it joins the network, moving load away from others to redistribute the load. This contrasts with our algorithm, which only rebalances on node departures and not on node joins. This turns out to be provably inefficient compared to an offline algorithm:

Theorem 4.7 *Any strategy which restores an even distribution of load after nodes join and leave has an unbounded competitive ratio.*

Proof. We give an example sequence with unbounded competitive ratio. Suppose n nodes are sharing some load L such that each has load $\frac{L}{n}$. A new node joins the system, and the load is redistributed by moving $\frac{L}{n+1}$ load to the new node. The new node then departs, returning the $\frac{L}{n+1}$ load to the original n nodes before doing so. This process can be repeated, with the original n nodes remaining connected. An optimal algorithm would leave the load at the original n nodes, and does not need to send any messages. □

This theorem can be trivially extended to include any strategy which moves any load to a new node as soon as it joins.

5 Full persistence

Full persistence in real-world situations requires the network to tolerate *ungraceful* node departures as well as graceful. For the sake of simplicity, we will only briefly discuss the changes which allow our system to be fully robust.

Replace each node in the SPON data structure (Fig. 3) with a cluster of nodes connected in a complete graph; replace each edge with a full bipartite graph between the two node clusters. Assuming that each node fails with constant probability $\alpha \in (0, 1)$, a cluster of size $O(\log n)$ is sufficient to say that with high probability no entire cluster will fail. The information assigned to each node can be stored in every node in the cluster, so that if a single node remains the information is not lost. Clusters depleted by node failures request additional nodes from the supervisor, which removes entire clusters from the active heap (and replaces the cluster data in the system) and uses the nodes contained in them.

If one assumes that an optimal offline algorithm would also need to keep $O(\log n)$ copies of each data item, it is easy to extend Theorem 4.5 to show that the load balancing algorithm based on node clusters is still $O(\log n)$-competitive. To preserve the competitiveness of the analysis, nodes used to refill clusters in the active heap must be drawn from the active heap, though in practice using nodes from the backup heap may be preferable since they are not currently storing any information.

6 System Design and Implementation

6.1 Design assumptions

We initially target our system design towards a small set of low-powered mobile domain nodes such as sensors, cellular phones, or PDA's connecting in an ad-hoc network. As a consequence of this, domain nodes cannot afford the battery power or computational effort required to use public-key cryptography, and simpler hash-based methods for authentication must be used. Specifically, our system uses cryptographic mechanisms based on Message Authentication Code (MAC) functions, such as HMAC [6] based on the hash function SHA-1 [23]. Given some input x and key k, $H_k(x)$ represents the outcome of applying the MAC function H_k to x.

We assume that the domain nodes share a key K_D which is only known to them. If the group of domain nodes is static, the key K_D can be stored on each node during the system setup. In scenarios where the group of nodes is dynamic, group key agreement protocols that support dynamic groups (such as CLIQUES[35, 2] coupled with a group communication system) can be used to compute the group key K_D.

6.2 Cryptographic protocols

We assume that every message has the following form:

source ID	dest ID	payload	security info

Registration. Before a visitor can get access to a domain, it has to receive a secret key from the domain. For instance, a visitor v may receive a key $K_v = H_{K_D}(ID(v))$, where $ID(v)$ is v's identifier; since K_D is shared by the domain nodes any one can provide such a key. This key will be used by the visitor to authenticate its messages and compute other secret keys. Notice that this registration phase is performed only once (as long as the key K_D does not change) and can thus be performed via traditional public-key based techniques; in particular, the use of a public-key certificate to prove $ID(v)$ keeps a visitor from acquiring multiple identities from the perspective of the domain.

Authentication. After registering, a visitor v has to authenticate itself to a domain node d by proving ownership of the identifier. To achieve this, it sends out a message of the form:

$ID(v)$	$ID(d)$	MID	$Time$
$H_{K_v}(ID(v), ID(d), MID, Time)$			

where MID is a random number used as message ID to uniquely identify the transaction. The use of the time information precludes replay attacks. Since d knows K_D, it can compute K_v and therefore immediately authenticate v.

Once d has verified that the message sent by v is correct, it checks whether v has already authenticated itself with d (not too long ago). If not, then d sends a message[3]:

$ID(d)$	$ID(v)$	MID	$Time$
$H_{K_D}(ID(d), ID(v), MID, Time)$			

to the supervisor (or to any node which has a copy of the blacklist, or to any node which is capable of deciding access permission). If the supervisor responds negatively, d will not give v the permission to access the domain.

If v's authentication is valid and v has permission to access the domain, d acknowledges this to v and becomes its guard (if it has not already done so). Afterwards, both d and v compute $H_{K_v}(ID(v), ID(d), Time)$ and store it as the key $K_{v,d}$. $K_{v,d}$ will be used by v and d to communicate with each other.

Communication between visitor and guards. Suppose that a visitor v wants a message M to be transmitted to node w via domain node d. In this case, it sends out the message:

$ID(v)$	$ID(w)$	M	MID	$Time$
$H_{K_{v,d}}(ID(v), ID(w), M, MID, Time)$				

[3]If confidentiality is required, the message can be first encrypted and the MAC can be computed over the resulting ciphertext

The domain node d computes the MAC function and verifies that the message is coming, intact, from the node v. Any domain node that is not a guard of v would reject this message.

The node d would then transform the message into:

$ID(v)$	$ID(w)$	M	MID	$Time$
$H_{K_D}(ID(v), ID(w), M, MID, Time)$				

to inform other domain nodes that the message has been authenticated and it is valid so that it can travel through the domain network without being dropped. As noted before, if confidentiality is required, the message can be first encrypted and the MAC can be computed over the resulting ciphertext.

Suppose now that a message M is sent to visitor v by some node w. If before reaching v the message arrives at a domain node d that is already a guard of v, then d will forward the message to v. Otherwise, d asks v to authenticate itself, using the authentication scheme described earlier. If this is successful, d will become a guard and forward the message to v.

If the traffic from or to a visitor node ceases for a certain period of time, the corresponding guard nodes may decide to deny the visitor node access to the network and change their status accordingly (requiring reauthentication with the next communication).

Sharing identities. As long as secret keys are never revealed, malicious nodes cannot take over the identity of any legitimate visitor. However, once receiving a key $K_{v,d}$ a malicious visitor v may share its identity with other unauthorized nodes. If time stamps are monitored and transmissions are concurrent, this can be easily detected by the domain nodes. If transmissions are not concurrent, then the group of visitors may simply be treated as one by the system. No monitoring system can prevent visitors from frequently exchanging their identities; see [18, 33] for discussions of this issue.

6.3 Implementation notes

It is important to remark that the design above cannot be extended to work with a large set of domain nodes. Indeed, techniques based on symmetric cryptography do not usually scale well unless the network has some specific structure. Also, assuming that each domain node stores the same key may be troublesome as it would be enough to compromise one single node to recover such a key.

Nevertheless, it works well for a small set of domain nodes and we want to minimize the use of techniques based on public key cryptography. In addition, it simplifies the creation of a prototype implementation of the system, which is currently in progress.

As first step, we have implemented one the most important components of our system, the SPON network. It has been implemented as a Java socket program that builds an overlay network based on UDP unicast primitives. Currently, the SPON network has been successfully tested on a network of Sun Ultra 5 (UltraSPARC-IIi 360 MHz) workstations connected over a 100baseT LAN; a small test network performed node join and leave operations and broadcast message passing with no visible delay.

The current usage of SPON's implementation is a broadcast-based "chat" program, built as a proof-of-concept and test vehicle for the network. For space considerations, a screenshot and explanatory details are in an appendix.

7 Acknowledgments

Many thanks to Joey Chau for implementing and setting up the SPON network.

References

[1] J.P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical Report, James P. Anderson Co., Fort Washington, Pennsylvania, April 1980.

[2] G. Ateniese, M. Steiner, and G. Tsudik. New Multiparty Authentication Services and Key Agreement Protocols. IEEE Journal on Selected Areas in Communication, 2000.

[3] A. Bakker, E. Amade, G. Ballintijn, I. Kuz, P. Verkaik, I. van der Wijk, M. van Steen, and A. Tanenbaum. The Globe distribution network. In *Proceedings of the 2000 USENIX Annual Conference (FREENIX Track)*, pp. 141-152, 2000.

[4] J.S. Balasubramaniyan, J.O. Garcia-Fernandez, D. Isacoff, E. Spafford, and D. Zamboni. An architecture for intrusion detection using autonomous agents. Technical Report 98/05, Purdue University.

[5] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pStore: A secure peer-to-peer backup system. Technical Report, MIT Laboratory for Computer Science, December 2001.

[6] M. Bellare, R. Canetti, and H. Krawczyk, Keying hash functions for message authentication. In *Proc. of Crypto 96*, 1996.

[7] J. Cannady and J. Harrell. A comparative Analysis of Current Intrusion Detection Technologies. In 4th Technology for Information Security Conference (TISC'96), May 1996.

[8] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital Libraries*, pp. 28–37, 1999.

[9] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, 2000. `http://freenet.sourceforge.net`.

[10] L.P. Cox and B.D. Noble. Pastiche: making backup cheap and easy. In the Fifth USENIX Symposium on Operating Systems Design and Implementation, December 2002, Boston, MA.

[11] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In Proceedings of the 18th ACM Symposium on Operating Systems Principles. October 2001.

[12] S. Elnikety, M. Lillibridge, M. Burrows, and W. Zwaenepoel. Cooperative backup system. In the USENIX Conference on File and Storage Technologies, Monterey, CA, January 2002.

[13] M. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*, 2nd edition. John Wiley and Sons, 2001.

[14] J. Hochberg, K. Jackson, C. Stallings, J.F. McClary, D. DuBois, and J. Ford. Nadir: An automated system for detecting network intrusion and misuse. *Computers & Security*, 12(3), 1993.

[15] S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith. Implementing a distributed firewall. In *Proceedings of Computer and Communications Security*, ACM CCS 2000.

[16] R. Janakiraman, M. Waldvogel, Q. Zhang: Indra: A peer-to-peer approach to network intrusion detection and prevention. In Proc. 2003 IEEE WET ICE Workshop on Enterprise Security, Linz, Austria, June 2003

[17] A.K. Jones and R.S. Sielken. Computer System Intrusion Detection: A Survey. University of Virginia, Computer Science Technical Report 2000.

[18] R. A. Kemmer. NSTAT: A Model-based Real-Time Network Intrusion Detection System. University of California-Santa Barbara Technical Report TRCS97-18, Nov. 1997.

[19] S. Kumar and E. Spafford. An Application of Pattern Matching in Intrusion Detection. Purdue Technical Report CSD-TR-94-013, June 1994.

[20] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pp. 190–201, 2000.

[21] G. Liepens and H. Vaccaroo. Intrusion Detection: Its role and Validation. Computer & Security 11 (1992) 347-355.

[22] T.F. Lunt. A Survey of Intrusion Detection Techniques. Computer & Security 12 (1993) 405-418.

[23] National Institute for Standards and Technology (NIST). *Secure Hash Standard.* FIPS 180-1, 1993. `http://www.itl.nist.gov/fipspubs/fip180-1.htm`

[24] Ohaha, Smart decentralized peer-to-peer sharing. `http://www.ohaha.com/design.html`

[25] G. Pandurangan, P. Raghavan, and E. Upfal. Building low-diameter peer-to-peer networks. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS '01)*, 2001.

[26] P.A. Porras and P.G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In 19th National Information System Security Conference (NISSC), 1997.

[27] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335-348, 1989.

[28] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage. IEEE Internet Computing, 5(5):40-49, Sept. 2001.

[29] C. Riley, C. Scheideler. Guaranteed Broadcasting Using SPON: Supervised Peer Overlay Network. To appear in the 2004 International Zurich Seminar on Communications. Available online as technical report, Johns Hopkins University, 2003. `http://www.cs.jhu.edu/~chrisr/papers/spon_tr.ps.gz`

[30] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In IFIP/ACM International Conference on Distributed Systems Platforms. Germany, Nov. 2001.

[31] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP '01)*, pp. 188–201, 2001.

[32] S.E. Smaha and J. Winslow. Misuse Detection Tools. Computer Security Journal 10.1 (1994) 39-49.

[33] S.R. Snapp, J. Brentano, G.V. Dias, T.L. Goan, L.T. Heberlein, C. Ho, K.N. Levitt, B. Mukherjee, S.E. Smaha, T. Grance, D.M. Teal, and D. Mansur. DIDS (Distributed Intrusion Detection System)-Motivation, Architecture, and An Early Prototype. In Proceedings of the 14th National Computer Security Conference, October 1991.

[34] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS-A Graph Based Intrusion Detection System for Large Networks. In 20th National Information System Security Conference (NISSC), October 1996.

[35] M. Steiner, G. Tsudik and M. Waidner. Key Agreement in Dynamic Peer Groups. IEEE Transactions on Parallel and Distributed Systems, 2000.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2001)*, ppp. 149–160, 2001.

[37] Murray Turoff. Past and Future Emergency Response Information Systems. In Communication of the ACM, April 2002/Vol. 45, No.4. Pages 29-32.

[38] G. White, E. Fisch, and U. Pooch. Cooperating security managers: A peer-based intrusion detection system. IEEE Network, 10(1), 1994.

Appendix: SPON screenshot

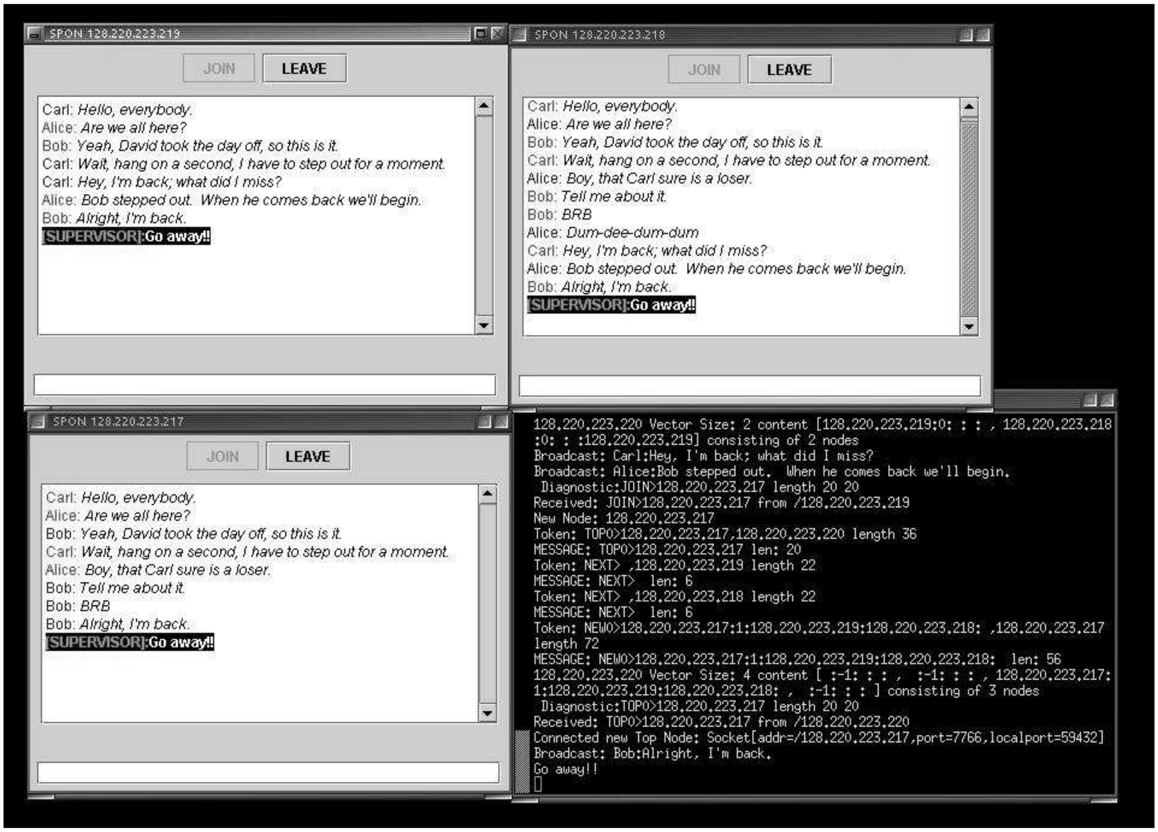

Figure 4. A screenshot of the SPON network in operation.

In the screenshot, the three similar windows correspond to separate peers being executed by remote terminals operating through SSH; these windows show the current SPON GUI. The lower right window is the supervisor window, which is text-only. In the peer GUI, the bottom field is for text input. Typing text and pressing “enter” will broadcast the text to all other peers and the supervisor. The “join” and “leave” buttons join and leave the network, respectively; since only one is possible at a time, the other is grayed out. The screenshot represents an arbitrary sequence of communications between Alice (218), Bob (217), Carl (219), and the Supervisor (220), interspersed with join and leave operations by Bob and Carl.