Routing in the Dark: Pitch Black

A Thesis
Presented to
the Faculty of Engineering and Computer Science
University of Denver

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Nathan S. Evans
June 2009
Advisor: Christian Grothoff

Author: Nathan S. Evans

Title: Routing in the Dark: Pitch Black

Advisor: Christian Grothoff

Degree Date: June, 2009

Abstract

In many networks, such as mobile ad-hoc networks and friend-to-friend overlay networks, direct communication between nodes is limited to specific neighbors. Friend-to-friend "darknet" networks have been shown to commonly have a small-world topology; while short paths exist between any pair of nodes in small-world networks, it is non-trivial to determine such paths with a distributed algorithm. Recently, Clarke and Sandberg proposed the first decentralized routing algorithm that achieves efficient routing in such small-world networks.

Herein this thesis we discuss the first independent security analysis of Clarke and Sandberg's routing algorithm. We show that a relatively weak participating adversary can render the overlay ineffective without being detected, resulting in significant data loss due to the resulting load imbalance. We have measured the impact of the attack in a testbed of 800 and 400 total nodes using minor modifications to Clarke and Sandberg's implementation of their routing algorithm in Freenet. Our experiments show that the attack is highly effective, allowing a small number of malicious nodes to cause rapid loss of data on the entire network.

We also discuss various proposed countermeasures designed to detect, thwart or limit the attack. We found that the "darknet" topology limits the ability of effective countermeasures. The problem of fixing the topology proved so intractable due to inherent network characteristics that the idea of using a darknet for Freenet has *been all but abandoned following the public release of this work. Our hope is that the presented analysis acts as a step towards effective analysis and design of secure distributed routing algorithms for restricted-route topologies.*

It should be noted that this thesis is an extended version of the same work presented at ACSAC 2007. The work appears in the conference proceedings as "Routing in the Dark: Pitch Black" [19] largely unmodified from this thesis.

Acknowledgments

I would like to thank my advisor, Christian Grothoff, Chris GauthierDickey and Ramki Thurimella for their help on the work presented here as well as with the writing of the original paper. I would also like to thank Katie Haus for her patience and help throughout the entire process of doing this research in innumerable ways.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Fully decentralized and efficient routing algorithms for restricted route networks promise to solve crucial problems for a wide variety of networking applications. Efficient decentralized routing is important for sensor and general wireless networks, peer-to-peer overlay networks and theoretically, next generation Internet (IP) routing. A number of distributed peer-to-peer routing protocols developed in recent years achieve scalable and efficient routing by constructing a structured overlay topology [11, 20, 34, 43, 56]. However, none of these designs are able to achieve these goals in real-world networks with *restricted routes*. In a restricted route topology, nodes can only directly communicate with a subset of other nodes in the network. Although many P2P (peer-to-peer) networks [8, 11, 14, 17, 34, 56, 61] make the assumption that any peer can communicate directly with any other peer, this is generally untrue in actuality. Such restrictions can arise from a variety of sources, such as physical limitations of the communications infrastructure (wireless signals, physical network topology), network policies (firewalls) or limitations of underlying protocols; for example NAT [4]. The most common of these is NAT (network address translation) which is widely used by home Internet service and business networks alike and makes arbitrary direct communication virtually impossible. Specifically, a computer that is behind a NAT can communicate with a non-NAT'ed computer as long as the NAT'ed peer initiates the communication. However the reverse is not true unless the NAT box has been configured to allow incoming communications (such as by port forwarding or otherwise). Similarly two NAT'ed peers are typically unable to communicate with each other without some intermediary set up to perform forwarding. Techniques for dealing with NAT have been proposed and implemented [46, 47, 55], but they are not perfect solutions, and not ubiquitous enough to solve the restricted routing problem that NAT creates. This prevalence of NAT makes applications unable to assume universal connectivity, even though this was the main problem that the Internet's routing infrastructure was created to solve (that any two peers can communicate) but unfortunately we must deal with the networks as they are and not as we would like them to be.

Recently, a new routing algorithm for restricted route topologies was proposed [50] and implemented in version 0.7 of Freenet, an anonymous peer-to-peer file-sharing network [7]. The proposed algorithm achieves routing in expected $O(\log n)$ hops for small-world networks with n nodes and $O(\log n)$ neighbors[1] by having nodes swap *locations* in the overlay under certain conditions. This significant achievement raises the question of whether the algorithm is *robust* enough to become the foundation for the large domain of routing in restricted route networks. Any peer-to-peer network operating on the Internet (or really any network) must be able to cope with malicious and byzantine nodes in the overlay that for whatever reason do not follow the correct protocol. For our purposes we will consider Freenet 0.7 "robust" if the inclusion of peers that outwardly appear to follow the protocol (but actually do not) are limited in the harm that they can do to the network.

[1]Given only a constant number of neighbors, the routing cost increases to $O(\log^2 n)$.

The research presented in this paper shows that any participating node can severely degrade the performance of the routing algorithm by changing the way it participates in the location swapping aspect of the protocol. Most of the guards in the existing routing implementation are ineffective or severely limited and in particular fail to reliably detect the malicious nodes. Experiments using a Freenet 0.7 testbed show that a small fraction of malicious nodes can dramatically degenerate routing performance and cause massive content loss in a short period of time. Our research also illuminates how churn impacts the structure of the overlay negatively, a phenomenon that was observed by Freenet 0.7 users in practice but had never been adequately explained by the time that we finished our research.

The paper is structured as follows. Chapter 2 describes related work focusing on distributed hash tables and small-world networks. Chapter 3 details Freenet 0.7's distributed friend-to-friend (or, as termed by the Freenet authors, "darknet") routing algorithm for small-world networks. The proposed attack is described in Chapter 4, followed by experimental results showing the effects of the attack in Chapter 5. Possible defenses and their limitations are discussed in Chapter 6.

Chapter 2

Related Work

In this section we first describe DHT's (distributed hash tables) and some of the P2P networks that were created based upon them. Freenet 0.7 is an implementation of a DHT so it is therefore important to understand a bit about them before introducing how Freenet 0.7 works. Background on other DHT systems is also useful for comparison with Freenet 0.7. Two things that are lacking from the basic DHT designs are the ability to route in restricted route topologies and security. We touch on these topics in this section as well, as Freenet's goals include both.

2.1 Distributed hash tables

A distributed hash table is a data structure that enables efficient key-based lookup of data in a peer-to-peer overlay network. Generally, the participating peers maintain connections to a relatively small subset of the other participants in the overlay. Each peer is responsible for storing a subset of the key-value pairs and for routing requests to other peers. In other words, a key property of the use of DHTs in a peer-to-peer setting is the need to route queries in a network over multiple hops based on limited knowledge about which peers exist in the overlay network. Part of the DHT protocol definition is thus concerned with maintaining the structure of the network as peers join or leave the overlay.

All data in a DHT is identified by some semi-unique value (this uniqueness usually comes from the use of a hash function, such as SHA or MD5, meaning that while collisions are possible they are very unlikely) which is commonly referred to as a *key*. This unique value comes from the large distribution of values that are possible outcomes of the hashing function. For an idea of how large, SHA-0 and SHA-1 have 2^{64} possible values. All peers or nodes in the network are also identified by a unique value from the same distribution known as the node's location or identifier. We use the term location and identifier interchangably through this thesis. In this way all nodes and data are separately and uniquely identifiable in the network. DHT's support two logical operations; a *PUT* operation (which inserts data identified by some key into the DHT) and a *GET* operation (which given a key returns data from the network).

There are some common properties that DHT's designs can be categorized by. One property which is a concern of all modern DHT designs is efficiency. Efficiency is defined in terms of the number of nodes a message must be routed to or through before the data that is being looked for is found. This measure is referred to as the number of "hops" a query must travel before data is found. There are two kinds of reliability that DHT designs take into account. The first is the reliability of the overall network; that it is able to remain operational in the presence of adversaries. The second is a matter of keeping data in the network reliably. This concerns distributing data across the network so it cannot be lost unless all nodes go down, or that data remains in the network as nodes go offline (which is a matter of replication). DHT's are also good at load balancing, or the distribution of responsibility. This spreads the costs of storing and retrieving data in the network across all nodes that are participating, which could mean more total load could be handled or no single node has to bear the whole burden. Finally, DHT designs should take into account scalability which means that the network retains its properties of efficiency, reliability, etc. for different total numbers of nodes.

Efficiency in a DHT is directly tied to the routing algorithm, or the steps that each node carries out in order to perform GET or PUT requests. There are two main types of routing algorithms that are employed; iterative routing and recursive routing. In iterative routing the node that initiates the query controls where the next hop goes at each step. In recursive routing the message is forwarded by intermediate nodes and the initiator only receives (or does not receive) the final reply. For a simple example of these two types of routing, assume three nodes, a, b and c. Assume that a knows b, b knows c and a is searching for data stored at node c. In iterative routing, node a contacts b requesting the data, b responds to a that it doesn't have the data, but knows another node, c and sends to a c's contact information. Node a then contacts c directly and finds the data. In recursive routing, node a contacts b with the same request, but b forwards the request directly to c, with the same result. So in iterative routing the path from node a to node c goes $a \rightarrow b \rightarrow a \rightarrow c$ and in recursive routing goes $a \rightarrow b \rightarrow c$.

2.1.1 Skip Lists

Skip lists are basically sorted linked lists in which items have additional pointers which allow much more efficient lookup of items in the list [41]. When searching a normal ordered linked list, unlike an ordered array where a binary search can be used, every node must be iterated over in order to verify the presence of an item. However, if additional pointers are stored at each entry in the linked list that point to nodes further forward than the next immediate node the total search time can be greatly reduced. This means that if a node in the linked list is at position i it normally only holds a pointer to $i+1$ (and $i-1$ if doubly linked). In a skip list the node at position i holds additional pointers to nodes at positions $i+n, i+k, ...$ Skip lists can give lookup costs of $O(\log n)$ based on how many additional pointers are stored, which allows DHT's an easy way of achieving the same average performance when routing is based on these augmented linked list structures.

The simplest DHT implementation based on skip lists is a circular linked list where each element in the linked list is stored at a different physical node (and each node is running on a different host computer). If each node only had a "pointer" to the next node (where by pointer we mean the required information to connect over the network to the DHT software) we would have a DHT that could be used as a circular linked list. Of course, storing a single value per node and requiring $O(n)$ steps to find the value would probably be a waste of effort. So this simple idea is extended by having multiple values stored at each node, and by giving each node extra connections to other nodes in the linked list. By using these extra connections we can essentially use a skip list over a network. In this way we can see how this example DHT and skip lists are virtually the same thing, and DHT's such as [11, 56] are simply extensions of this idea.

2.1.2 Scalable Content Addressable Network

The first DHT described herein is the Scalable Content Addressable Network (CAN)[43]. CAN imposes structure onto the unstructured underlying network by assigning **zones** in some d-dimensional coordinate space where a zone is a rectangle, rectangular prism and so forth depending on the number of dimensions. The diagram in Figure 2.1 shows a 2-d coordinate space although any number of dimensions are usable. In a 2-d space, the plane is divided into zones where each node n is responsible for at least one such zone. Being responsible for a zone means that a node will store any data that hashes to a point in that zone. The neighbors of a node n are the nodes which overlap in one axis and abut in another (in our 2-d space). The CAN is constructed by nodes splitting their current zones in half to allow new nodes to join. As described earlier for DHT's in general, each node and data item in CAN is identified by a random location, in the case of CAN it is some point in the coordinate space.

The construction of CAN starts from a single node, which has a location and is initially responsible for the entire coordinate space (a single zone encompassing the entire space). When the second node joins, the first splits its zone in half, then notifies the new node of the space for which it will be responsible. This happens whenever a new node enters the CAN; it generates a random point in the coordinate space, and some other node that is currently responsible for that point splits itself apart to give the new node a zone. The new node's routing table is filled from information given by the split node, as it necessarily knows the adjacent nodes of the new space. The split node simply removes those nodes that it no longer needs and adds the new node to its routing table. Routing in CAN is done recursively, when a *GET* or *PUT* request comes into a node (either from another node or a client application), the node projects a line from its own location to the location of the request destination and forwards the message on to the first neighbor that the projected line travels through. In this way, the message eventually reaches the node responsible for the zone in which the point lies and can respond to the query, or store the data. Figure 2.1 shows the nodes and selected neighbor relationships for nodes in a 2-d space in order to illustrate these ideas. The example also shows the path that routing through the CAN takes based on each node choosing the neighbor that is first encountered by drawing a line from the current node to the destination.

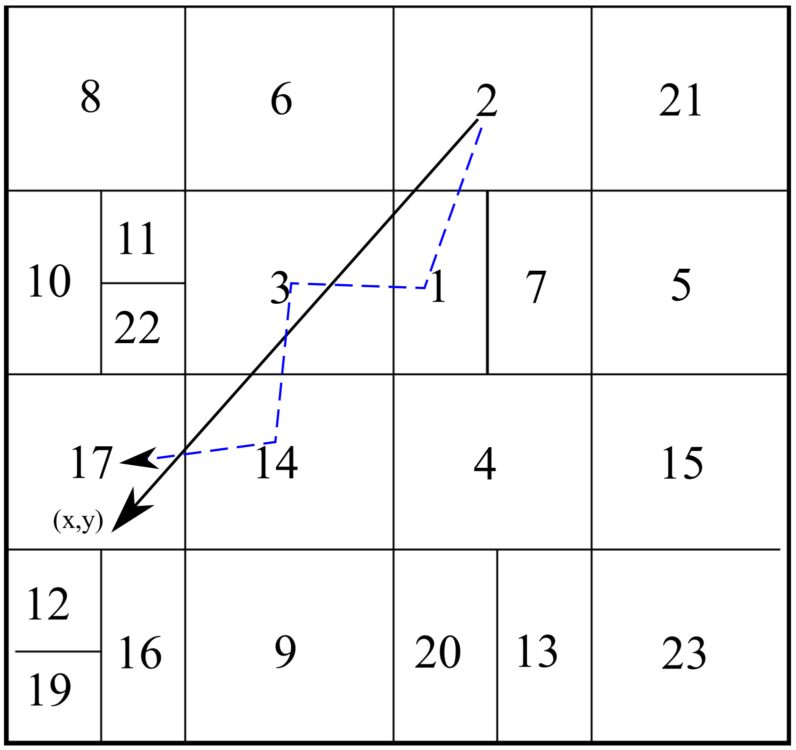


Figure 2.1. This figure shows an example of how nodes could be laid out in a CAN network. The full neighbor set for node 2 is $\{6, 21, 1, 7, 20, 13\}$. Remember that $20, 13$ are neighbors because the space is structured as an n-d torus (with $n = 2$). Also shown (in dashed lines) is the nodes along the routing path that results when node 2 searches for data which hashes to point (x, y). The solid line shows an example of how node 2 chooses which node to initially route to.

Provided that there are no byzantine nodes in the CAN network the CAN address space is partitioned evenly on average between all the nodes in the network. Nodes that depart gracefully transfer their zone to either a node which can merge with the old zone and make a valid zone, or the neighbor node with the smallest volume temporarily takes over the old zone. A background maintenance protocol runs that checks for node liveness, and if a node is discoverd to no longer be online (or reachable), the smallest neighbor or a neighbor that can make a new valid zone from the old takes over the zone left behind. The background protocol also runs at each node to try to find “misshapen” zones and repartition them into valid zones in the coordinate space. CAN as described has an average routing cost of $O(n^{-d})$ messages. Due to the even partitioning of the address space based on zones and linking storage to the zones gives CAN scalability, because as more nodes are added the zones get smaller and spread the storage cost equally across all nodes. Allowing for multiple CANs to coexist and for more than one node to “share” a zone can give CAN good reliability and replication in the face of node departures. CAN does degrade to flooding the network with expanding ring queries when too many nodes leave in a short period, which means that CAN is not the best system for networks with lots of nodes joining and leaving frequently.

2.1.3 Kademlia

Kademlia was mainly created to improve upon previous DHT designs that were available at the time. The The biggest difference between Kademlia and these earlier DHT designs [48, 56] is that it uses the XOR operation to determine distance between elements in the key space. This means that given two nodes in the network the *distance* between any two nodes a, b is given by the value of the two XOR'ed together $D_{a,b} = L(a) \oplus L(b)$ where $L(x)$ is the location of node x. Pastry uses two different metrics for deciding how close keys in the address space are to each other (as discussed in 2.1.5). One of Kademlia's goals is to unify multiple distance metrics into a single metric that can be used for routing from start to finish. As in all early DHT designs, a Kademlia node is assumed to be able to connect to any other *live* node in the system at any time. Kademlia achieves its routing performance from the property that all the keys in the address space can be ordered into a binary tree. Binary trees are known to have $O(\log n)$ search times and this allows Kademlia to locate keys in the address space in a short time, provided that the routing tables are sufficiently populated.

Kademlia's routing table is structured as a certain number of so called k-buckets, of which there are as many k-buckets as there are bits in the locations of nodes (i.e. a 160 bit location means 160 k-buckets per node). Each k-bucket can hold an arbitrary (but set by a parameter) number of triples which contain the necessary information to connect to another node, specifically the IP address, port, and location of the node. For each k-bucket i, the node stores other nodes information whose locations are between 2^i and 2^{i+1} distance from the storing node's location. These buckets get populated whenever a node receives a query from any other node. When a query is received the querying node's pertinent information is stored in the k-bucket that corresponds to its location on the side of the node receiving the query. The k-buckets are maintained such that the least recently seen node will get bumped out first if the bucket is full and another node that belongs in the same bucket is encountered (but only if the least recently seen is unresponsive, otherwise the new node information is discarded). Routing in Kademlia is iterative; there is no forwarding of queries between nodes. To understand how Kademlia nodes find keys in the address space it is important to know how the routing tables are constructed. When a node first joins the network it creates its own random location. A node that is joining the network must have knowledge of at least one other node's IP address and port a priori. The new node connects to whatever node or nodes it knows. This node may have any location, and the distance to the new node can be any number in the address space. Once the new node has connected to at least one peer it sends a request out to its peer(s) to find out what other nodes in the network exist that have locations closest to the new node's location. The distance is determined by the metric mentioned above. Assume that this new node, z, has only one pre-known peer to connect to, x. Once z has generated its random location, it connects to x asking for any and all nodes (up to some limit parameter j set by each node) that x knows of that are closest to z's location. The first thing that x does is check its k-buckets and insert z if necessary. Then x responds to z with the closest j nodes that x knows of (or less than j if x has less than j total nodes in all its k-buckets). z then adds all the nodes it has learned about from x to its k-buckets appropriately. Now z can repeat this process and query any of the new peers it has discovered for *their* closest j nodes to its location until it has either enough nodes total in its k-buckets or until it has as many within a certain distance of its own location as specified as a system or user parameter. The same node discovery process is employed when a node performs a *GET* request. The node takes a key that is to be searched for (based on hashing or some arbitrary method) and then queries j peers closest to the key's value based on those that it knows about (those that are stored in its k-buckets). Each peer that receives this query either responds with its j closest peers to the key or with a "data found" response (if it happens to be holding the data). Routing in Kademlia achieves good performance in part due to the binary tree structure inherent in the graph. Proving the effectiveness of the routing in Kademlia requires one major assumption, that if we divide the binary tree that makes up the Kademlia graph into first two sub-trees divided at the root (into node locations starting with a 0 or 1), then in the half that is different from a particular node's location, the node will have at least one contact in that sub-tree. Put simply, a node whose first identifier bit is 0 is guaranteed to have a node in it's routing table whose first identifier bit is 1. Further, if we traverse one node down towards the actual nodes location and split in half again, any node is again guaranteed to have a contact in the sub-tree not containing its own location.

Figure 2.2 shows how this division looks in a concreate visual example of a binary tree. Once these connections are assumed to exist it is easy to see that whatever key a node is looking for, it will be able to find a node half as close as itself by prefix (at least) in a single query to a node in the correct sub-tree. Again, this node is assumed to exist in the routing table. Figure 2.3 shows an example of the queries involved and the routing in Kademlia.

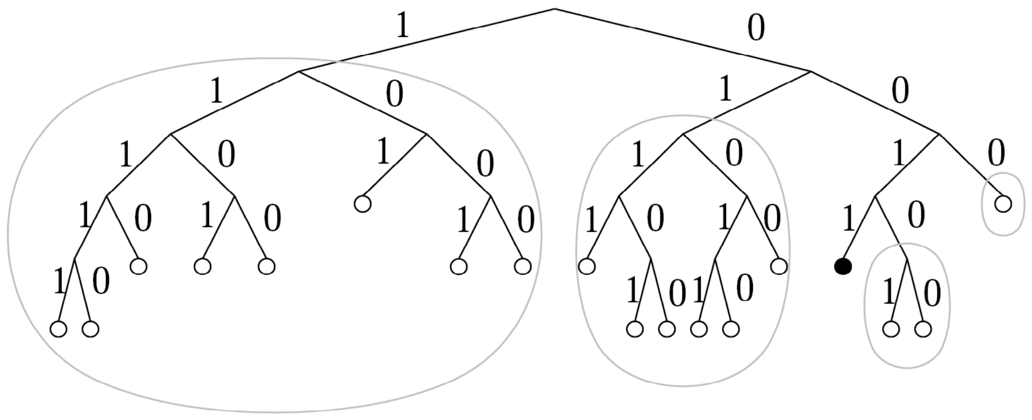


Figure 2.2. This figure shows the binary tree representing nodes in a Kademlia topology. The node in black is the node we are looking at, and the gray circles represent sections of the graph in which this node is assumed to have a contact. (From [34])

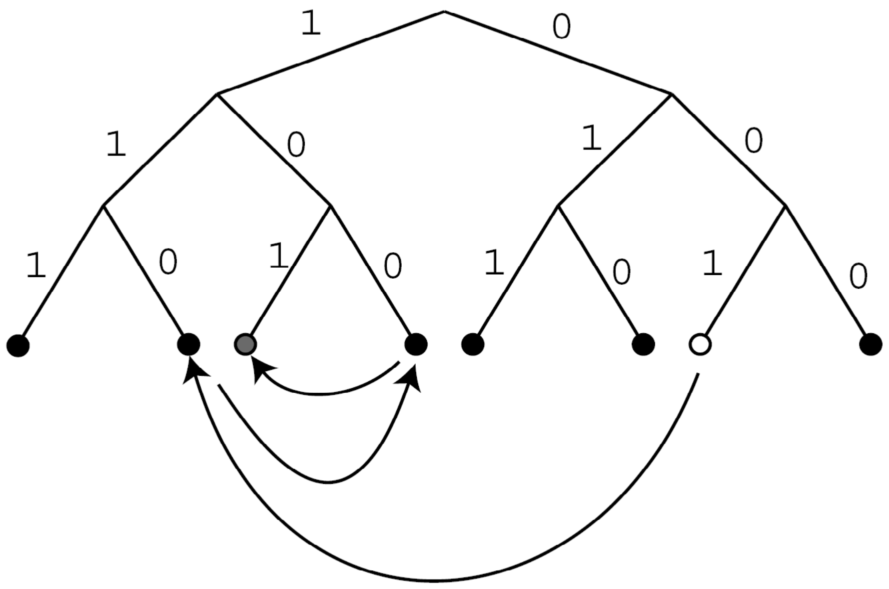


Figure 2.3. This figure shows the steps a node takes when searching for a key in the Kademlia topology. The node in white (identified by 001) is searching for the node in gray (101). The first query from 001 goes to node 110 (which shares the prefix 1 with 101), which then sends the node information for node 100 (which shares the prefix 10 with the target), which then sends back the node information for 101 back to the original node and the connection can happen directly between 001 and 101. It is easy to see that by increasing the matching prefix size by 1 bit the address space is cut in half at each step. (From [34])

2.1.4 Chord

Chord is an early DHT system that was developed to be simpler in terms of routing and routing table maintenance than previous systems yet achieve nearly the same performance. Chord's topology is most easily explained using points plotted on a circle. Assuming that a node location can be mapped to some point on the circle, the node maintains routing information for the next immediate node on the circle in clockwise direction (its successor) and for the next immediate node in counter-clockwise direction (its predecessor). All nodes are responsible for any data whose key lies on the circle between the node's location and the location of its predecessor. Therefore routing is a straightforward forwarding in a clockwise direction until the node that is responsible for the data is found, i.e., the key identifying the request lies between a node and its predecessor. Routing in Chord is done recursively, and nodes are expected to forward requests on in clockwise order as they are received. The topology is maintained as nodes join and leave the network by simply shifting the predecessor and successor node information at each node, as would be done in a doubly linked list. The routing table for each node keeps information for peers that are further away than their successor to achieve better routing performance in skip list fashion (Chord is literally an implementation of a distributed skip list). Similarly, when a node leaves unexpectedly (without informing its neighbors) there are steps that need to be taken, basically by periodically sending out keepalive messages to make sure neighbors are live. If a neighbor is discovered to have gone offline, the next peer along the circle must be found to replace it in the routing table. Figure 2.4 shows an example set of nodes in a Chord topology to help illustrate the predecessor and successor relationships between nodes.

2.1.5 Pastry

Pastry is another early DHT peer-to-peer system. Pastry uses network locality as a way to help refine its routing table entries. Pastry builds a similar topology to other systems detailed here but arguably has the most complicated system for maintaining its list of neighbor nodes or routing table(s). As with other DHT's, each Pastry node has a random location that it uses to identify itself. Similar to Kademlia 2.1.3, Pastry uses prefix matching when searching for close locations in the network. Pastry uses recursive routing, so nodes in Pastry are expected to greedily forward queries to the

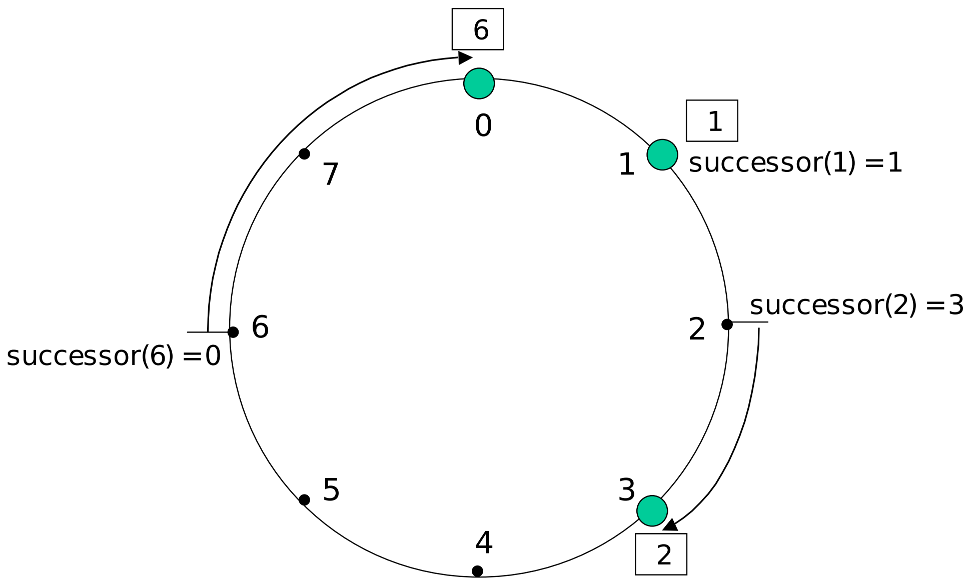
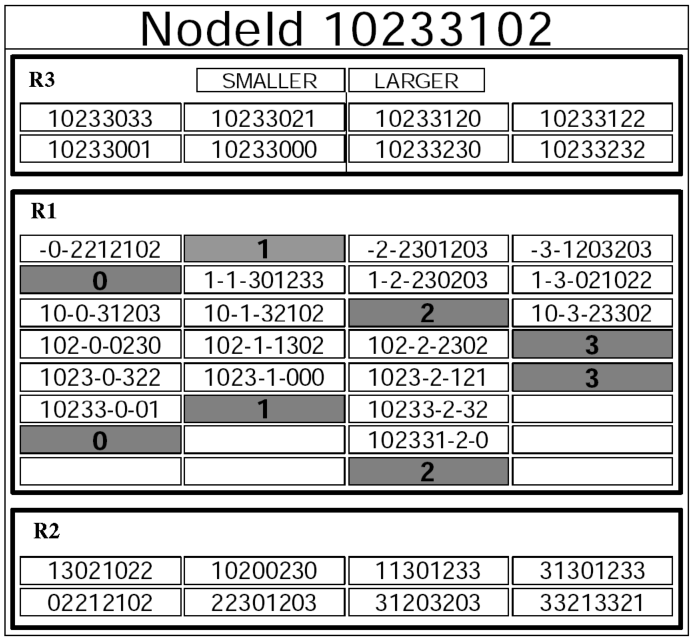


Figure 2.4. This figure shows three nodes in a Chord topology along with their simple predecessor ($pred(x) =$ node location that is the predecessor of node with location x) and successor ($succ(x) =$ node location that is the successor of node with location x) relationships

neighbor whose location has a longer matching prefix than its own location. However, Pastry uses the numerical difference between locations for forwarding when a query reaches the point that a node has no neighbor with a longer matching prefix than itself.

Pastry maintains a routing table that is made up of three parts, a *routing table* (R1), a *neighborhood set* (R2) and a *leaf set* (R3). The R1 set of neighbors is the largest part of the routing table, with $\log_{2^b} N$ rows where b is a system wide parameter and N is the number of nodes. Finding a good estimate of the number of nodes in the system at a given time is difficult in itself, but there are methods which attempt to do so [23, 33]. Each row of R1 has $2^b - 1$ possible entries, where the i^{th} row has entries of peers that have a location with the first $i + 1$ digits matching the node's location. This table is built when the node enters the network by (assuming a connection to one node a priori) issuing a special join query which, based on the greedy routing algorithm, will reach the nearest current node in the system. All nodes along the path send back their complete routing tables to the new node and based on matching prefix length and some network proximity metric (IP routing hops as described for the original version of Pastry) is used to build the R1 part of the routing table. The R2 and R3 sets are created in a similar manner to the R1 set. However, the R2 set contains m peers (where m is a system or configuration parameter) that are *closest* in terms of the locality metric that is being used. The R3 set contains the closest m nodes whose locations are greater than the node's location, and the m closest nodes whose locations are less than the node's location. The R3 set is the first set used in the greedy routing algorithm, followed by the R1 set if the key being searched for does not fall within the range of values in the R3 set. The R2 set is used only for maintaining locality measures and is used when deciding which nodes to keep in the R1 set.

The R2 set is not used for updating either the R1 or the R3 sets directly; rather if two nodes are in contention for a place in R1, and one of them is closer in proximity (i.e. it is in the neighborhood set) the closer is chosen to hold that spot in the table. Pastry assumes that a locality metric is available which can give a measure of how close or far a node is according to some measure. In this way after building the R1, R2, and R3 tables, both R1 and R2 are continuously checked based on this proximity metric to ensure that the locality properties hold. This is necessary because network locality metrics such as routing hops are not necessarily transitive. A node builds its initial R2 and R1 sets from nearby nodes (and they should report only their closest nodes), sometimes locality is different for the new node. In other words, if a node X is close to Y, and Y is close to Z, it is likely but not always the case that X is also close to Z. Or X could be closer to Z than Y. In either case, the node X verifies these locality measures to make sure that the routing tables stay up to date and correct. Thus the Pastry overlay is constructed. Figure 2.5 depicts an example of the entire routing table for a Pastry node in a network of 4-bit identifiers, and Figure 2.6 shows only the R1 portion of a different node in a network with 16-bit identifiers, only in a more intuitive manner which may make it easier to see how neighbors are ordered.


NodeId 10233102

R3

	SMALLER	LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

R1

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

R2

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure 2.5. This figure shows the routing table for the node with the location 10233102. R1 shows the closest nodes based on matching prefix, R2 shows the closest nodes in terms of locality (IP hops) and R3 shows the closest nodes based on numerical distance.

2.1.6 Secure DHT's

Security in routing is another problem that needs to be dealt with in P2P networks. In the networks described in this section security is generally not a consideration. But ideally a network must be able to guarantee a route around and/or find data in the presence of faulty nodes, possibly discern between good and bad information that is

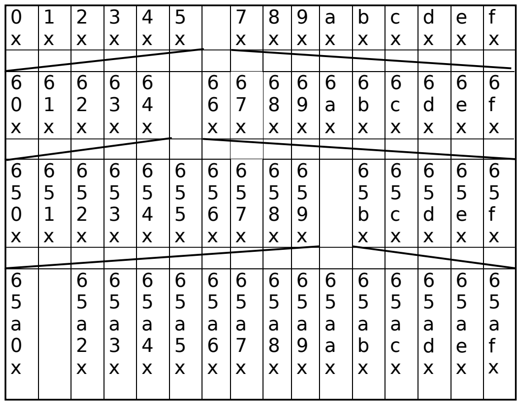

0x 1x 2x 3x 4x 5x 7x 8x 9x ax bx cx dx ex fx
60x 61x 62x 63x 64x 66x 67x 68x 69x 6ax 6bx 6cx 6dx 6ex 6fx
650x 651x 652x 653x 654x 655x 656x 657x 658x 659x 65bx 65cx 65dx 65ex 65fx
65a0x 65a2x 65a3x 65a4x 65a5x 65a6x 65a7x 65a8x 65a9x 65aax 65abx 65acx 65adx 65aex 65afx

Figure 2.6. This figure shows only the R1 portion of a pastry routing table for a node whose location begins with 65a1 as it shows more clearly how each level of the routing table contains more "close" entries based on prefix.

present in the network, and have a reasonably random distribution of node identifiers (locations). Also the network protocol should not allow peers to use a larger amount of CPU or bandwidth than they provide to the network. There are examples of these types of attacks (and possible solutions) for most of the DHT networks in use today.

A typical way to "secure" a DHT is to assume that a trusted third party exists that can perform operations for the peers in the overlay. This trusted third party would perform operations such as distributing node locations, possibly public key cryptography information and determining malicious behavior and banning nodes from the network. Unfortunately a trusted third party is rare come by in the world of P2P networks, due to the high cost (either monetary or CPU/bandwidth) of using one. P2P file sharing networks based on DHT's are free, and users are unwilling to pay for a third party. A Byzantine agreement protocol [30] could be used to create a virtual trusted third party but agreement requires prohibitively large communication overhead. Achieving routing robustness (routing around faulty/malicious nodes) and guaranteed storage is typically accomplished by randomizing routes along multiple paths and replicating data at multiple peers as well. Randomizing routes means that identical queries can be forwarded to different nodes; as long as a correct peer is forwarding the query it will continue towards the correct destination [12]. Similarly, replicating data at multiple nodes in the network thwarts attempts at deleting, modifying or blocking access to any piece of information. In many networks replication is non-trivial because sybil [15] attacks must be considered. A sybil attack is a type of attack where a node in a network can imitate or run a group of colluding nodes and perform some of the attacks that replication is meant to prevent. For example, if nodes can choose their own identifiers in the network and replication is not sufficiently randomized, a group of nodes whose identifiers are clustered around a desired data item can still control that piece of data. Sybil attacks can also partition networks and control requests coming from specific peers in the same way.

One way to ensure data remains in a network is to use some form of BFT (Byzantine Fault Tolerant) replicated data storage to maintain state in the system [36]. The main problem with this type of system is that it can handle only up to $\frac{1}{3}$ of faulty nodes. Most P2P systems have very little protection against one malicious adversary impersonating many nodes in the system. Without a trusted third party this remains a problem for a BFT type of solution. Redundant or randomized routing is a rather good solution to the problem of finding routes around malicious (or byzantine) nodes in the network, but it comes at the cost of efficiency. Generally it takes more messages and hops to route past bad nodes, although the tradeoff may be worth it considering otherwise data may not be found. While several papers have looked at the problems inherent in DHT security and proposed solutions [5, 12, 27, 49, 53] the issues are truly only mitigated at best, and secure routing remains an open problem.

(1)	Messages required for each key lookup
(2)	Messages required for each store operation
(3)	Messages needed to integrate a new peer
(4)	Messages needed to manage a peer leaving
(5)	Number of connections maintained per peer
(6)	Topology can be adjusted to minimize per-hop latency (yes/no)
(7)	Connections are symmetric or asymmetric

Table 2.1. Performance metrics for DHTs.

	Chord [56]	Pastry [48]	Kademlia [34]	CAN [43]	RSG [20]
(1)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n^{-d})$	$O(\log n)$
(2)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n^{-d})$	$O(\log n)$
(3)	$O(\log^2 n)$	$O(\log n)$	$O(\log n)$	$O(d + n^{-d})$	$O(\log n)$
(4)	$O(\log^2 n)$	$O(1)$	$O(1)$	$O(d)$	$O(\log n)$
(5)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(d)$	$O(1)$
(6)	no	yes	yes	yes	no
(7)	asymmetric	asymmetric	symmetric	symmetric	asymmetric

Table 2.2. Comparison of DHT designs. The numbers refer to the list of performance metrics given in Table 2.1. The value d is a system parameter for CAN.

2.1.7 Performance Comparison

DHT designs can be characterized using the performance metrics given in Table 2.1. Routing in DHTs is generally done in a greedy fashion and resembles lookups in skip lists [41]. Table 2.2 summarizes the key properties of various existing DHT designs. The table does not capture properties which are hard to quantify, such as fault-tolerance. Given a uniform distribution of keys, most existing DHT designs achieve near perfect load balancing between peers. Hosts that can provide significantly more resources than others are usually accommodated by associating multiple locations in the overlay with a single host. In some sense, those hosts are counted as multiple peers.

A major limitation of the DHT designs listed in Table 2.2 is that they do not support routing in restricted route topologies. These DHTs assume that it is generally possible for any peer to connect to any other peer. However, firewalls and network address translation (NAT) make this assumption unrealistic over the current Internet, where large-scale studies have shown that over 70% of machines are NAT'ed [4].

In contrast to the DHT designs from Table 2.2, the Freenet 0.7 routing algorithm achieves expected $O(\log n)$ routing in restricted route topologies under the assumption that the restricted network topology has small-world properties. The Freenet 0.7 algorithm also attempts to take into account failed nodes and some types of malicious attacks, although these measures are not the focus of our research.

2.2 Small-world networks

A small-world network is informally defined as a network where the average shortest path between any two nodes is "small" compared to the size of the network, where "small" is generally considered to mean at least *logarithmic* in relation to the size of the network. Small world networks occur frequently in the real world [59], the most prominent example being social networks [37].

Watts and Strogatz [59] characterized small-world networks as an intermediate stage between completely structured networks and random networks. According to their definition, small world networks with n nodes have on average k edges per vertex where $n > k > \log n$. They define a *clustering coefficient* which captures the amount of structure (clustering) in a given network. Small-world networks are then networks with a clustering coefficient significantly larger than the coefficients of completely random networks and with average shortest path lengths close to those of completely random networks. Watts and Strogatz's work explains why short paths exist in real-world networks.

Kleinberg [28, 29] generalized Watts and Strogatz' construction of small-world networks and gave sufficient and necessary conditions for the existence of efficient distributed routing algorithms for these constructions. Kleinberg's model for distributed routing algorithms does not include the possibility of nodes swapping locations, which is a fundamental part of Freenet 0.7's "darknet" routing algorithm.

Chapter 3

Freenet's "darknet" routing algorithm

Freenet [7] is a peer-to-peer network where the operator of each node specifies which other peers are allowed to connect to the node [6]. The main reason for this is to obscure the participation of a node in the network – each node is only directly visible to the friends of its' operator. Peer-to-peer networks that limit connections to friend-to-friend interactions are sometimes called *darknets*. Given that social networks are small-world networks and that small-world networks arise easily given a certain amount of "randomness" in the graph construction, it is realistic to assume that Freenet 0.7's darknet is a small-world network. The routing restrictions imposed on the Freenet 0.7 overlay could technically model arbitrary network limitations; consequently, an efficient distributed routing algorithm for such a topology should easily generalize to any small-world network.

3.1 Network creation

The graph of the Freenet 0.7 network consists of vertices, which are peers, and edges, which are created by friend relationships. An edge *only* exists between peers if both operators have agreed to the connection a priori. Freenet 0.7 assumes that a sufficient number of edges (or friend relationships) between peers will exist so that the network will be connected.

Each Freenet 0.7 node is created with a unique, immutable *identifier* and a randomly generated initial *location*. The identifier is used by operators to specify which connections are allowed, while the location is used for routing. The location space has a range of $[0, 1)$ and is cyclic with 0 and 1 being the same point. For example, the distance between nodes at locations 0.1 and 0.9 is 0.2.

Data stored in the Freenet 0.7 network is associated with a specific *key* from the range of the location space. The routing algorithm transmits GET and PUT requests from node A to the neighbors of A starting with the neighbor with the closest location to the key of the request.

3.2 Operational overview

The basic strategy of the routing algorithm is to greedily forward a request to the neighbor whose location is closest to the key. However, the simple greedy forwarding is not guaranteed to find the closest peer – initially, the location of each peer is completely random and connections between peers are restricted (since a peer can only establish connections to other peers which the operator has explicitly allowed). Consequently, the basic greedy algorithm is extended to a depth-first search of the topology (with bounded depth) where the order of the traversal is determined by the distance of the nodes to the key [51]. Figure 3.1 shows the routing algorithm for GET operations in pseudocode. A PUT operation is routed in the same fashion and reaches exactly the same peers as an unsuccessful GET operation. In addition, Freenet 0.7 replicates content transmitted as part of a GET response or as part of a PUT operation at nodes that are encountered during the routing process where the node's location is closer to the key than the location of any of the peer's neighbors.

Both GET and PUT requests include a hops-to-live value which is initially set to the nodes pre-set maximum and used to limit traversal of the network. Each request also includes the closest location (in relation to the key) of any node encountered so far during the routing process.

Figure 3.1. Pseudocode for routing of a GET request.

```
1. Check that the new $GET$ request is not identical to recently processed requests;
   if the request is a duplicate, notify sender about duplication status, otherwise
   continue.

2. Check local data store for the data; if the data is found, send response to sender,
   otherwise continue.

3. If the current location is closer to the key than any previously visited location,
   reset hops-to-live to the maximum value.

4. If hops-to-live of the request is zero, respond with data not found, otherwise
   continue.

5. Find the closest neighbor (in terms of peer location) with respect to the key of
   the $GET$ request, excluding those routed to already. Forward the request to
   the closest peer with a (probabilistically) decremented hops-to-live counter. If
   valid content is found, forward the content to sender, otherwise, repeat step 5.
```

3.3 Location swapping

To make the routing algorithm find the data faster, Freenet 0.7 attempts to cluster nodes with similar locations. Let $L(n)$ denote the current location of node n. The network achieves this by having nodes periodically consider swapping their locations using the following algorithm:

1. A node A randomly chooses a node B in its proximity and initiates a swap request. Both nodes share the locations of their respective neighbors and calculate $D_1(A, B)$. $D_1(A, B)$ is the product of the existing distances between A and each of A's neighbors $|L(a) - L(n)|$ multiplied by the product of the existing distances between B and each of B's neighbors.

$$D_1(A, B) = \prod_{(A,n)\in E} |L(A) - L(n)| \cdot \prod_{(B,n)\in E} |L(B) - L(n)| \tag{3.1}$$

2. The nodes also compute $D_2(A, B)$, the product of the products of the differences between their locations and their neighbors' locations *after* a potential swap:

$$D_2(A, B) = \prod_{(A,n)\in E} |L(B) - L(n)| \cdot \prod_{(B,n)\in E} |L(A) - L(n)| \tag{3.2}$$

3. If the nodes find that $D_2(A, B) \leq D_1(A, B)$, they swap locations, otherwise they swap locations with probability $\frac{D_1(A,B)}{D_2(A,B)}$. The deterministic swap always decreases the average distances of nodes with their neighbors. The probabilistic swap is used to escape local minima.

The overlay becomes semi-structured as a result of swapping locations; the routing algorithm's depth first search can utilize this structure in order to find short paths with high probability. Sandberg's thesis [50] shows that the Freenet 0.7 routing algorithm converges towards routing in $O(\log n)$ steps (with high probability) under the assumption that the set of legal connections specified by the node operators forms a small-world network. This is a significant result because it describes the first fully decentralized distributed hash table (DHT) design that achieves $O(\log n)$ routing with (severely) restricted routes. Most other DHT designs make the unrealistic assumption that every node is able to directly communicate with every other node [20, 34, 48, 56].

3.4 Content Storage

Each Freenet 0.7 node stores content in a datastore of bounded size. Freenet 0.7 uses a least-recently-used content replacement policy, removing the least-recently-used content when necessary to keep the size of the datastore below the user-specified limit.

Our simulation of routing in the Freenet 0.7 network places the content at the node whose location is closest to the key and does not allow caching or replication of content. The reason for this is that our study focuses on routing performance and not on content replication and caching strategies.

3.5 Example

Figure 3.2 shows a small example network. Each node is labeled with its location ($L_n \in [0, 1)$) in the network. The bi-directional edges indicate the direct connections between nodes. In a friend-to-friend network, these are the connections that were specifically allowed by the individual node operators, and each node is only aware of its immediate neighbors. Similarly, in an ad-hoc wireless network, the edges would indicate which nodes could physically communicate with each other. While our example network lacks cycles, any connected graph is allowed; the small-world property is only required to achieve $O(\log n)$ routing performance but the algorithm will work (to some extent) for any connected graph.

The network illustrated in Figure 3.2 happens to have an assignment of locations

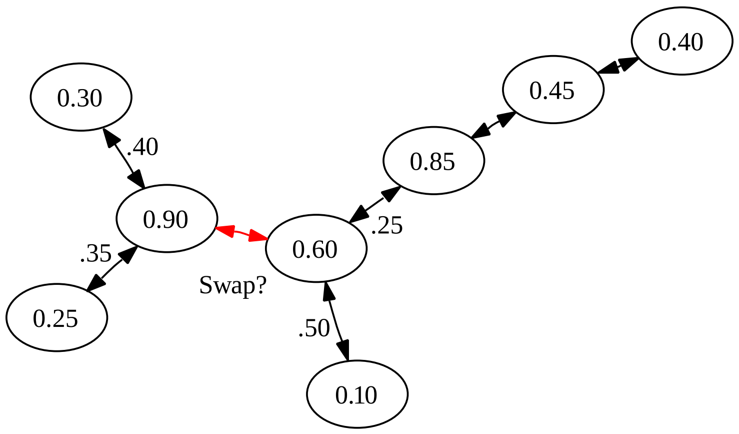


Figure 3.2. This figure shows an example network with two nodes considering a swap. The result of the swap equation is D_1 = .40 * .35 * .25 * .50 = .0175 and D_2 = .30 * .35 * .05 * .80 = .0042. Since $D_1 > D_2$, they swap.

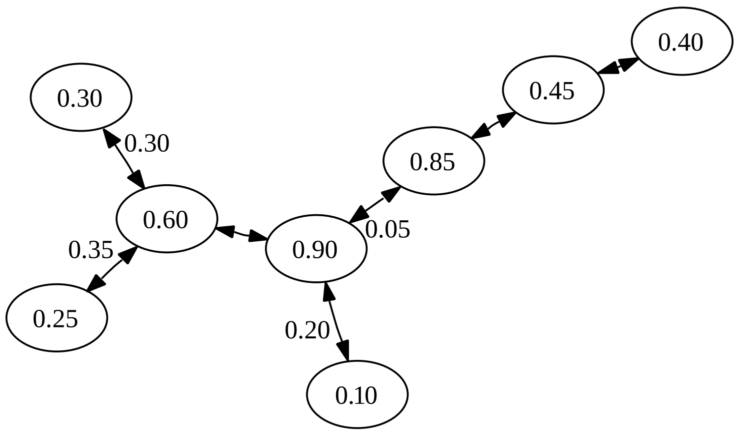


Figure 3.3. This figure shows the resulting example network after the swap has occurred.

that would cause the nodes with locations 0.60 and 0.90 to perform a swap in order to minimize the distance product from Equation (3.1). Figure 3.3 shows the new assignment of locations after the swap. Note that after a swap each node retains exactly the same set of connections; the only change is in the location identifiers of the nodes. This change in node locations impacts the order of the traversal during

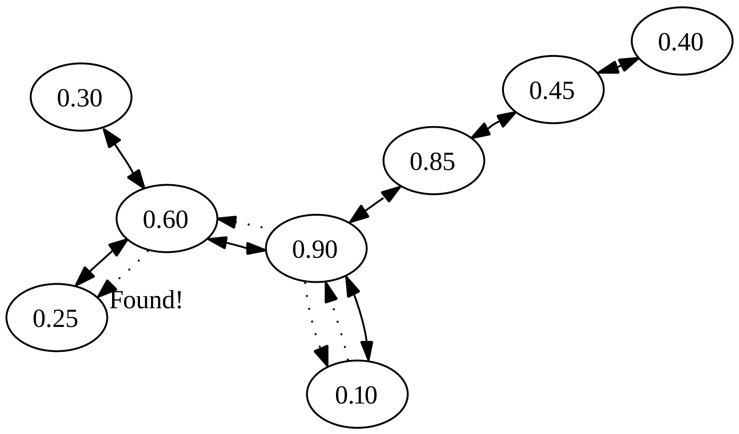
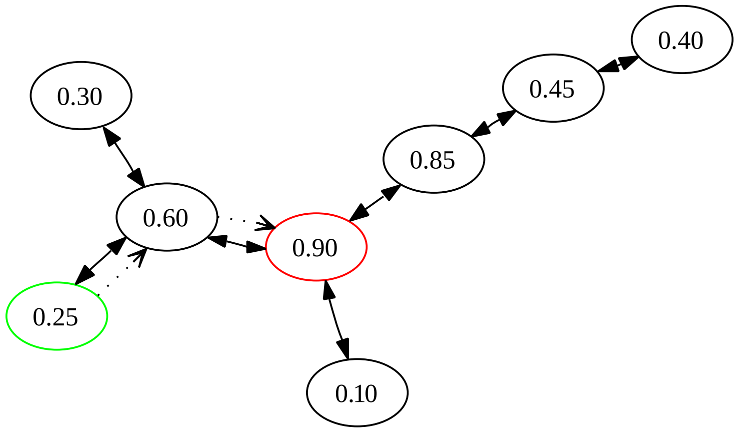


Figure 3.4. Illustrates the path of a GET request initiated from the node with location of 0.90. The request is looking for data with a key value of .23, which is stored at the node identified by the location 0.25. The path that the GET request travels is displayed as the dotted lines which travel from 0.90 $\rightarrow$ 0.10 $\rightarrow$ 0.90 $\rightarrow$ 0.60 $\rightarrow$ 0.25 where the data is found. The data reverses the successful path in order to make it back to the originator.

routing.

Figure 3.4 shows how a GET request would be routed after the swap (with a maximum value of hops-to-live larger or equal to two). Starting at the node with location 0.90 and targeting the key 0.23, the node picks its closest neighbor (with respect to the key), which is 0.10. However, 0.10 does not have the content and also lacks other neighbors to route to and thus responds with content not found. Then 0.90 attempts its' second-closest neighbor, 0.60. Again, 0.60 does not have the content, but it has other neighbors. The 0.25 neighbor is closest to 0.23. The content is found at that node and returned via 0.60 (the restricted-route topology does not allow 0.25 to send the result directly back to 0.90).

Finally, Figure 3.5 illustrates how Freenet 0.7 routes a PUT request with a maximum value of 1 for hops-to-live (in practice, the maximum value would be bigger). The algorithm again attempts to find the node with the closest location in a greedy fashion. Once a node C is found where all neighbors are further away from the node, the neighbors fail to reset hops-to-live (since 0.90 is closer to the key than they are), which ends the process.



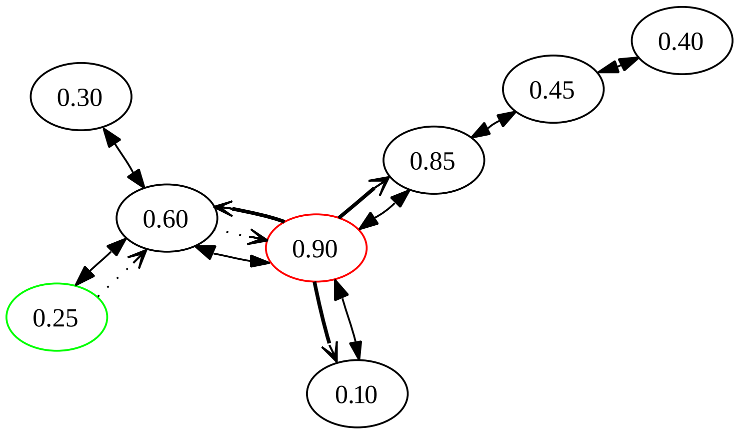


Figure 3.5. The graph on top illustrates the path of a PUT request inserting data with a key of .96. The request is initiated from node with location 0.25. The path that the PUT request travels is displayed as the dotted lines which travel from 0.25 $\longrightarrow$ 0.60 $\longrightarrow$ 0.90, where the data is stored. The bottom graph shows what happens after a PUT has found a node whose neighbors are all further away from the key. The node 0.90 (as all of the predecessors on the path) resets the hops-to-live value to its maximum (in this case, one) and forwards the PUT request to all of its neighbors. Since these neighbors are not closer to the key than their predecessor they do not reset hops-to-live. Since the value reaches zero, routing ends.

Chapter 4

Security Analysis

The routing algorithm works under the assumption that the distribution of the keys and peer locations is random and evenly distributed over the address space. In that case, the storage and routing load is evenly balanced. In particular, all nodes are expected to store roughly the same amount of content and all nodes are expected to receive roughly an equivalent numbers of requests. This key assumption is what gives Freenet 0.7 both the load balancing qualities that are desired in any DHT or P2P file sharing system, and is the main assumption that must be true for the algorithm to achieve $O(\log n)$ steps for routing in practice.

The basic idea behind the attack is to de-randomize the distribution of the node locations of the peers in the network. The attacker tries to cluster the locations around a particular small set of values. Since the distribution of the keys is still random and independent of the distribution of the node locations, the clustering of node locations around particular values results in an uneven load distribution. Nodes within the interior of the clusters are responsible for basically no content (because many other nodes are also close to the same set of keys), whereas the load for nodes on the outside edges of the clusters is disproportionately high.

We will now detail two scenarios which remove the vital randomness from the initial random distribution of node locations which results in the clustering of locations around particular values. The first scenario uses attack nodes which behave according to our malicious behavior as described in Section 4.1 inside the network. This attack quickly and effectively unbalances the load in the network, which causes significant data loss. The reason for the data loss is that the imbalance causes some nodes to greatly exceed their storage capacity, whereas other nodes store nothing. A node exceeds its storage capacity when the protocol dictates that particular node should store more data than it is configured for; a node faced with this scenario drops the least recently inserted data. At this point the data is lost, as no node in the network will be able to find the data even if the original inserter is still present. The second scenario illustrates how location imbalance can occur naturally even without an adversary due to churn (which also leads to overload and data loss).

4.1 Active Attack

As described in Section 3.3, a Freenet 0.7 node attempts to swap location identifiers with random peers periodically. Suppose that an attacker wants to bias the location distribution towards a particular location, m. In order to facilitate the attack, the attacker assumes that particular location (i.e. it sets its location to m). This malicious behavior cannot be detected by the node's neighbors because the attacker can claim to have obtained this location legitimately from swapping with some other node. A neighbor cannot verify whether such a swap has occurred because the friend-to-friend (F2F) topology restricts communication to immediate neighbors.

Suppose an attacker node A intends to force a swap with a victim N so that $L(N) = m$ after the swap occurs. Let N have k neighbors. Then A will initiate a swap request with N claiming to have at least $k+1$ neighbors with locations favoring a swap according to Equation (3.1). Specifically, the locations of the neighbors should be either close to $L(N)$ or close to the maximum distance from $L(A) = m$. The attacker then creates swap requests in accordance with the Freenet 0.7 protocol. Again, the F2F topology prevents the neighbor involved in the swap from checking the validity of this information. After the swap, the attacking node can again assume the original location m[1] and continue to try to swap with its other neighbors whose locations are still random.

The neighbors that have swapped with an attacker then continue to swap in accordance with the swapping algorithm, possibly spreading the malicious locations. Once the location has been spread, the adversary again subjects the neighbor to a swap, removing yet another random location from the network. Figure 4.1 illustrates the impact of a malicious node on the example network after a few swaps (with the attacker using $m \approx 0.5$). The likelihood of neighbors spreading the malicious location by swapping can be improved by using multiple attack locations, and switching between them periodically. Thus, a tradeoff exists between the speed of penetration and the impact of the attack in terms of causing load imbalances. For our experiments we typically chose speed over load imbalance because we were interested in having experiments that were relatively short lived (i.e. a few hours). Of course, in the real world network attackers could sit on the network for days or longer and likely cause even larger load imbalances than those we describe in 5.

4.2 Natural Churn

Network churn, the joining and leaving of nodes in the network, is a crucial issue that any peer-to-peer routing protocol needs to address. We have ignored churn until

[1]In practice malicious nodes use locations within small distance of m instead of exactly m in part to prevent the detection of this malicious activity

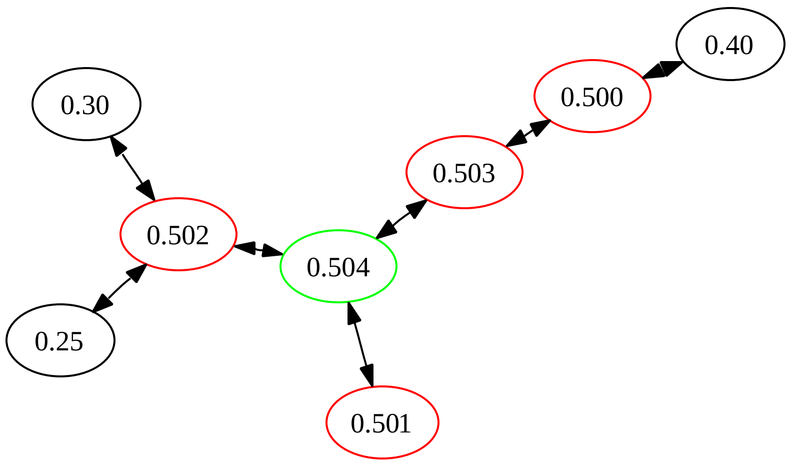


Figure 4.1. This figure shows the example network after a malicious node has started to spread locations close to 0.5 by swapping. In this figure the malicious node currently has location = 0.504

now because the attack described in the previous section does not require it. Intuition may suggest that natural churn may help the network against the attack by supplying a constant influx of fresh, truly random locations. This section illustrates that the opposite is the case: natural churn can strengthen the attack and even degenerate the Freenet 0.7 network in the same manner without the presence of malicious nodes.

For the purpose of this discussion, we need to distinguish two types of churn. The first kind, *leave-join churn*, describes fluctuations in peer availability due to a peer leaving the network *for a while* and then joining again. In this case, the network has to cope with a *temporary* loss of availability in terms of connectivity and access to content stored at the node. Freenet 0.7's use of content replication and its routing algorithm are well-suited to handle this type of churn. Most importantly, a node leaving does not immediately trigger significant changes at any other node. As a result, an adversary cannot use leave-join churn to cause significant disruption of network operations. Most importantly, honest Freenet 0.7 peers re-join with the same location that they last had when they left the network, leave-join churn does not impact the overall distribution of locations in the network.

The second kind, *join-leave churn*, describes peers who join the network and then leave *for good*. In this case, the network has to cope with the *permanent* loss of data stored at this peer. In the absence of adversaries, join-leave churn may be less common in peer-to-peer networks; however, it is always possible for users to discontinue using a particular application. Also, often users may just test an application once and decide that it does not meet their needs. Again, we believe that Freenet 0.7's content replication will likely avoid significant loss of content due to realistic amounts of join-leave churn.

However, natural join-leave churn has another, rather unexpected impact on the distribution of locations in the Freenet 0.7 overlay. This additional impact occurs when the overlay has a stable core of peers that are highly available and strongly interconnected, which is a common phenomenon in most peer-to-peer networks. In contrast to this set of stable core-peers, peers that contribute to join-leave churn are likely to participate only briefly and have relatively few connections. Suppose the locations γ_i of the core-peers are initially biased towards (or clustered around) a location $\alpha \in [0, 1)$. Furthermore, suppose that (over time) thousands of peers with few connections (located at the fringe of the network) contribute to join-leave churn.

Each of these fringe-peers will initially assign itself a random location $\beta \in [0, 1)$. In some cases, this random choice β will be closer to α than some of the γ_i-locations of the core nodes. In that case, the routing algorithm is likely to swap locations between these fringe-peers and core-peers in order to reduce the overall distances to neighbors (as calculated according to Equation (3.1)). Peers in the core have neighbors close to α, so exchanging one of the γ_i's for β will reduce their overall distances. The fringe peers are likely to have few connections to the core group and thus the overall product after a swap is likely to decrease.

Consequently, even non-adversarial join-leave churn strengthens any existing bias in the distribution of locations among the long-lived peers. The long-term results of join-leave churn are equivalent to what the attack from Section 4.1 is able to produce quickly – most peers end up with locations clustering around a few values. Note that this phenomenon has been observed by Freenet 0.7 users and was reported to the Freenet developers – who at the time when this research was originally published, had failed to explain the cause of this degeneration.[2] Since both the attack and natural churn have essentially the same implications for the routing algorithm, the performance implications established by our experimental results described in Section 5 hold for both scenarios.

[2]`https://bugs.freenetproject.org/view.php?id=647`, April 2007. We suspect that the clustering around 0.0 is caused by software bugs, resulting in an initial bias for this particular value, which is then strengthened by churn.

Chapter 5

Experimental Results

This chapter presents experimental results obtained from a Freenet 0.7 testbed with up to 800 active nodes. The testbed, consisting of up to 19 GNU/Linux machines, runs the actual Freenet 0.7 code. For the main results presented here, the nodes are connected to form a small-world topology (using Kleinberg's 2d-torus model [28]) with on average $O(\log^2 n)$ connections per node. As mentioned previously this is because we believe (as did the Freenet 0.7 authors/developers) that a small world network was likely to emerge from the "darknet" construction of the network. However, to be thorough we also experimented with some other topologies which have been commonly used in P2P networks. These topologies include: small-world networks without a 2d-torus, a 2d-torus only, a 2d-torus with "supernodes", and a small-world graph augmented with supernodes. Supernodes are common in some networks such as [17] and are very highly connected nodes in the network. The reason for the relatively small number of nodes (800 and 400) for our experiments is twofold. First, the estimated size of the actual Freenet 0.7 network based on open experimentation before we began our research was between 100 and 500 nodes. Therefore we feel that 800 and 400 nodes are good test sizes, as they are larger (or equal to) the size of the real network (at least when we began our research). The second reason is that we were bounded in our experiments by memory. Since Freenet 0.7 is a Java application we needed to use a Java virtual machine, and after some experimentation we chose the Sun JavaEE 1.6 SDK. Although we can limit the total amount of memory that the virtual machine could use, we found that each Freenet 0.7 node required about 64 MB to run smoothly. Since our lab machines are tasked for other purposes as well as ours, we found we could only run 50 Freenet 0.7 nodes per machine at any given time. Using 800 nodes computationally bounded us as well for simulating the routing that occurs in the Freenet 0.7 network, which meant that an 800 node test took greater than 5 hours. We found that we could do a test with 400 nodes in about one hour, and could try many more topologies and attacker configurations with the 400 node tests.

Each experiment consists of a number of iterations, where each iteration corresponds to 90 seconds of real time in the case of 800 node experiments, and 45 seconds for the 400 node experiments. In each iteration, nodes are given this amount of time in order to allow them to swap locations. Then the performance of the network is evaluated. The main performance metrics are the average path length needed to find the node that is responsible for a particular key not including dead end paths or failed searches, the average number of actual hops needed to terminate the query (including dead end paths), the average number of hops assuming that failed queries would reach all nodes, the number of “poisoned” locations, and the percentage of the content originally available in the network that can successfully be retrieved.

All nodes are configured with the same amount of storage space. Before each experiment, the network is seeded with content with a random key distribution. The amount of content is fixed at a quarter of the storage capacity of the entire network (unless otherwise specified). The content is always (initially and after each iteration) placed at the node with the closest location to the key. Nodes discard content if they do not have sufficient space. Discarded content is lost for the duration of the experiment.

Depending on the goals of the experiment, certain nodes are switched into attack mode starting at a particular iteration. The attacking nodes are randomly chosen, and behave exactly as all of the other nodes, except for aggressively propagating malicious node locations when swapping.

5.1 Distribution of Node Locations

Figures 5.1, 5.2, 5.3 and 5.4 illustrate the distribution of node locations on a circle before, during and after an attack. The initial distribution in Figure 5.1 consists of 800 randomly chosen locations, which are evenly distributed over the entire interval (with the exception of some noticeable clustering around 0.0, which we believe is due to implementation problems).

The distributions shown in Figures 5.2, 5.3 and 5.4 illustrate the effect of two nodes attacking the network in an attempt to create eight clusters around particular locations. Note that the number of attackers and the number of cluster locations can be chosen independently. We chose two attackers because it is believable that an adversary could run enough nodes to compromise .125 percent of all nodes given the small size of the actual network, and the choice of eight locations was because we found that to be a sufficient number for the attack to spread quickly. Figure 5.2 shows the node locations at an early point in the attack, after the 15^{th} attack iteration (90^{th} overall iterations. Even so, the clustering effect of our location swapping attack begins to be seen. In 5.3 the attack has been carried out for 75 iterations and the distribution of node locations can clearly be seen to be skewed towards the eight attacker chosen locations. Figure 5.4 shows the last picture of node locations at the very end of the attack; almost all nodes in the network have been forced to have an attacker chosen location. This example illustrates quite clearly how easy it is to remove the randomness necessary in node distributions in the Freenet 0.7 network. These results are also very typical, in all of the trials we performed with our attackers on the Freenet 0.7 testbed we saw nearly identical results.

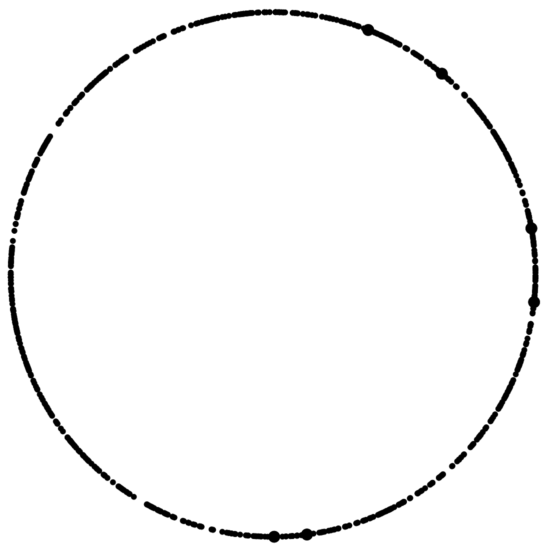


Figure 5.1. Plot of 800 initial node locations before the attack. Plot points increase in diameter as the density of peers nearby to that location increases.

All the plots use thicker dots in order to highlight spots where many peers are in close proximity. Particularly after the attack peers often have locations that are so close to each other (at the order of 2^{-30}) that a simple plot of the individual locations would just show a single dot. Thicker dots illustrate the number of peers in close proximity, the spread of their locations is actually much smaller than the thickness may suggest. The precise method for determining point size is as follows; each point as plotted on the circle has an x, y coordinate. The Cartesian distance from each

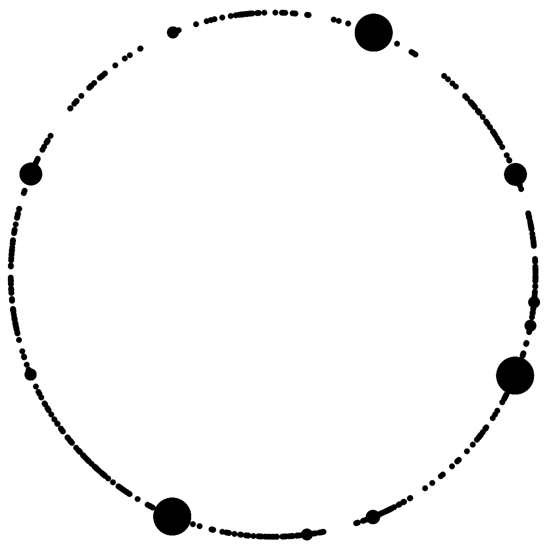


Figure 5.2. Plot of 800 initial node locations 15 iterations into the attack. Convergence to the attacker chosen locations begins to be apparent.

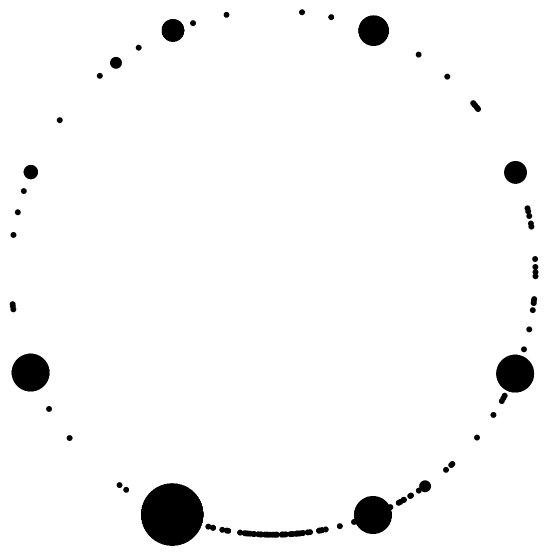


Figure 5.3. Plot of 800 initial node locations 90 iterations into the attack. Nearly all locations have been swapped into those chosen by the attacker at this point.

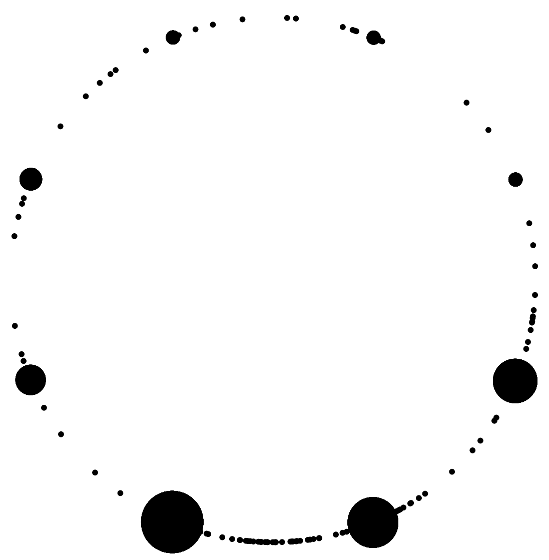


Figure 5.4. Plot of 800 node locations after the attack by 2 malicious nodes. The large plot points indicate the success of the attack in clustering most nodes around the 8 chosen locations.

point i to every other point $j \in S$ (where S is the set of all points), is calculated as $D_{i,j} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, and if that $D_{i,j}$ is less than some chosen amount, the points size value is incremented. To keep the points from growing out of control the $\log^2$ of each size value is taken as a multiplier applied to the base size of the points as plotted. To give some scale as to how close together two points need to be to increase the size of the displayed dot, the chosen threshold distance between points is $\frac{1}{100000000}$.

The second set of plots show the effectiveness of the attack in the same manner on the 400 node test network, this time using 16 malicious nodes and the attack nodes spreading only four locations. These results are impressive given that in this scenario the attack lasts only $1/4^{th}$ as long as the 800 node attack. Figure 5.5 shows the initial plot of node locations, again very evenly spread over the range of possible values. Figure 5.6 shows the picture after 25 iterations of the attacked network, and again the effect of the attack is readily apparent, and Figure 5.7, showing the result of 50 iterations of attack time tells the same story. Figure 5.8 shows the attack at the final attack measurement, 75 iterations after the attack began. Another way to see the effectiveness of the malicious swapping algorithm is shown in Figure 5.9, which plots the number of poisoned nodes against the run tested. It is clear that shortly after the attack begins, swaps are forced with all of the malicious nodes immediate neighbors. This accounts for the large jump near the 25^{th} iteration (when the attack is initiated). After this spike malicious node locations can only be spread by the normal swapping algorithm and therefore grows much more slowly, but the increase remains steady though the returns are diminishing. This is because the attacker needs to continue to iterate through all possible malicious locations until the neighbor of the forcibly swapped peer chooses to swap. As the attack location propogates futher from the malicious node it takes more time to iterate through all the attacker chosen locations.

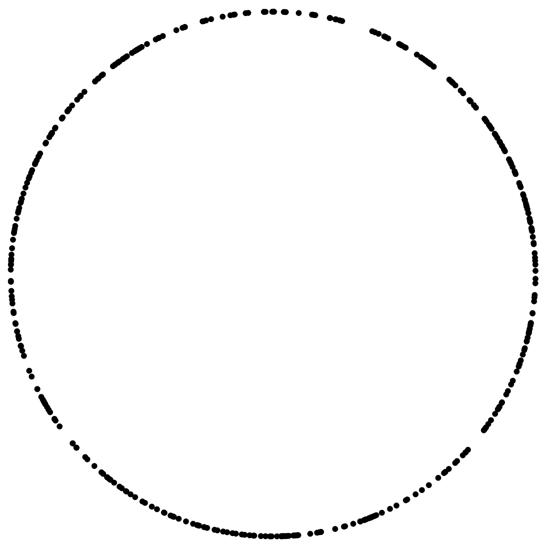


Figure 5.5. Plot of 400 initial node locations before the attack.

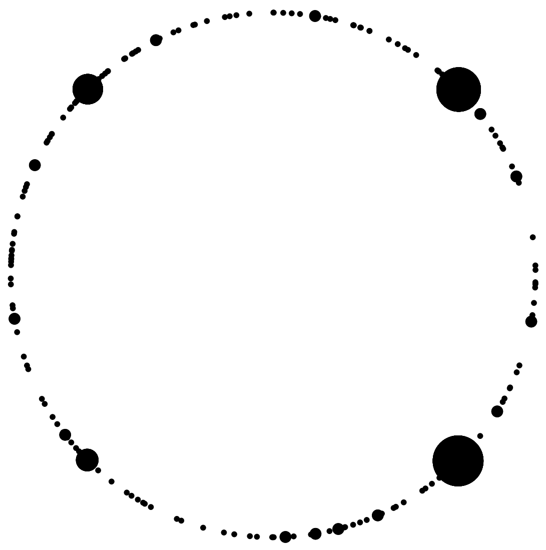


Figure 5.6. Plot of 400 node locations 25 iterations into the attack.

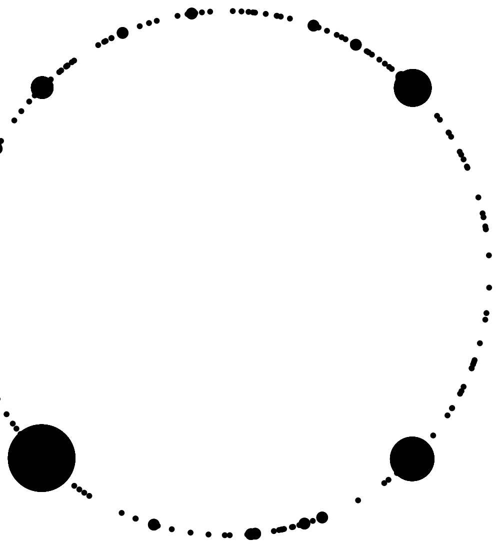


Figure 5.7. Plot of 400 node locations 50 iterations into the attack.

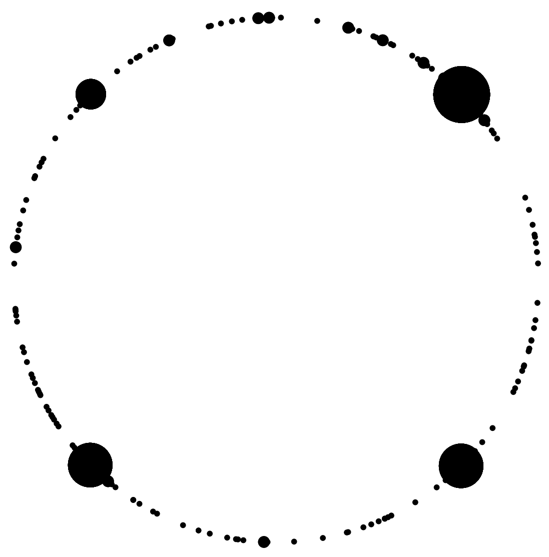


Figure 5.8. Plot of 400 node locations after the attack by 16 malicious nodes.

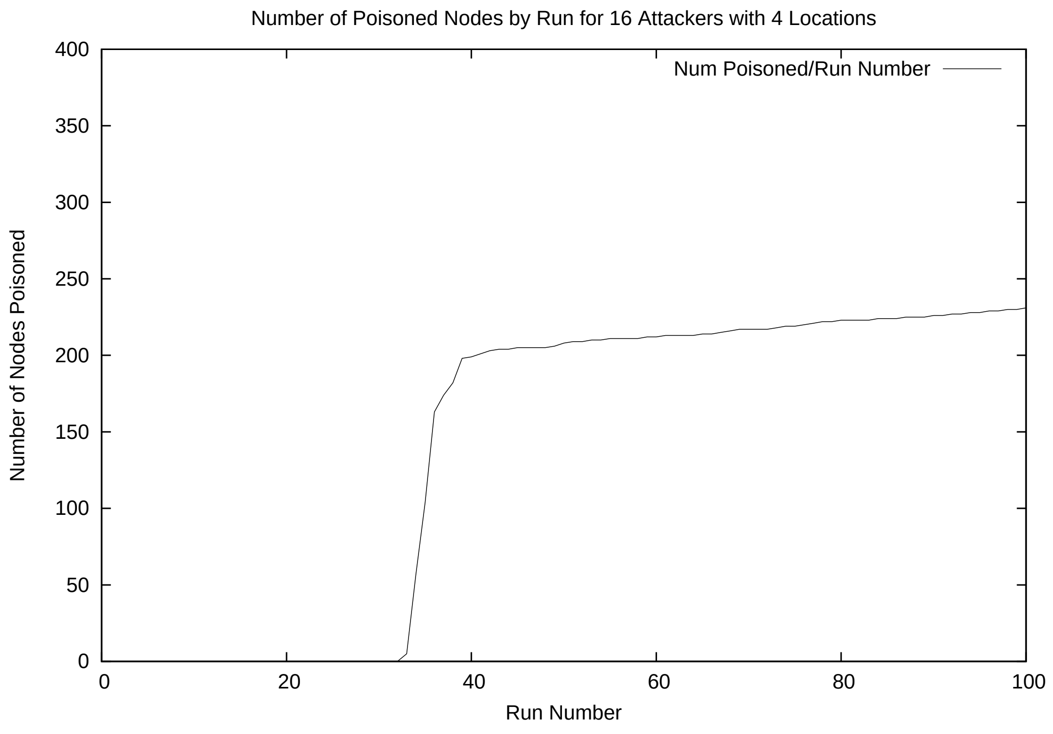


Figure 5.9. Plot of the number of nodes that have been “poisoned” by run number.

# Attackers	# Locations	Round 1	Round 50	Round 75	Round 100
1	2	0	29	29	39
1	4	0	31	56	67
1	8	0	32	60	69
1	16	0	23	48	80
2	2	0	44	51	59
2	4	0	57	66	75
2	8	0	77	113	137
2	16	0	73	120	145
4	2	0	67	75	86
4	4	0	79	99	114
4	8	1	110	139	155
4	16	0	139	197	224
8	2	0	101	112	121
8	4	0	157	169	178
8	8	0	194	211	225
8	16	0	192	244	256
16	2	0	113	174	183
16	4	0	205	215	231
16	8	0	239	253	266
16	16	0	272	289	299

Table 5.1. Data showing average number of poisoned nodes for varying configurations of our 400 node testbed and different points in time.

Table 5.1 shows the number of nodes that have been poisoned by attacking the network at different stages of the attack. The table displays the average number of bad nodes for varying numbers of attackers and node locations used in our 400 node testbed. The data is displayed for the first round, where there are no attackers, and then subsequent rounds 50, 75 and 100 to cover the full duration of the attacks. The obvious conclusion is that adding more attackers makes the attack work faster and that using more locations for a particular number of attackers generally also increases the number of nodes that take on poisonous locations. Round one is included because it is not a given that there are not always no nodes categorized as poisoned at the beginning of the attack. It is possible for a node to be randomly given a location which meets the criteria for being a “bad” location. However, this only happened in one of our experiments out of around 300 trials. It should also be noted that these averages are rounded up to the nearest whole number.

5.2 Routing Path Length

An important aspect of any routing algorithm is the number of hops that a message must travel before terminating, either in failure or in success. However there are a number of different ways to look at the hops that a message travels. In the Freenet 0.7 routing algorithm the maximum number of hops that a message can travel along a single path is capped at a constant of 10. For most of our hop counts we use this cap as well. One different metric for counting routing hops that we employ is a count of how many hops are traversed when route lengths are capped at a very high number (in our case this is unlimited). We used this measure because we thought it might give different numbers than the bounded hop count. Figures 5.10 show an example of this metric for our 400 node network where the attack begins at the 25^{th} iteration using 8 malicious nodes and 8 locations. The effect of the attack can clearly be seen in this plot, the number of hops needed for queries initially drop sharply as the network converges via the swapping algorithm. Once the attack is started the number of hops jumps back up, although never deteriorating to the length of the paths at the beginning of the experiment. We plot the mean of the averages over 5 runs in the graphs in this section, leaving out standard deviations for clarity.

Even using this max single path length still leaves some choices for measurements. As described in Chapter 3, if routing along one path fails then the next closest node is chosen until all nodes have been tried. Therefore there is the count of hops that are

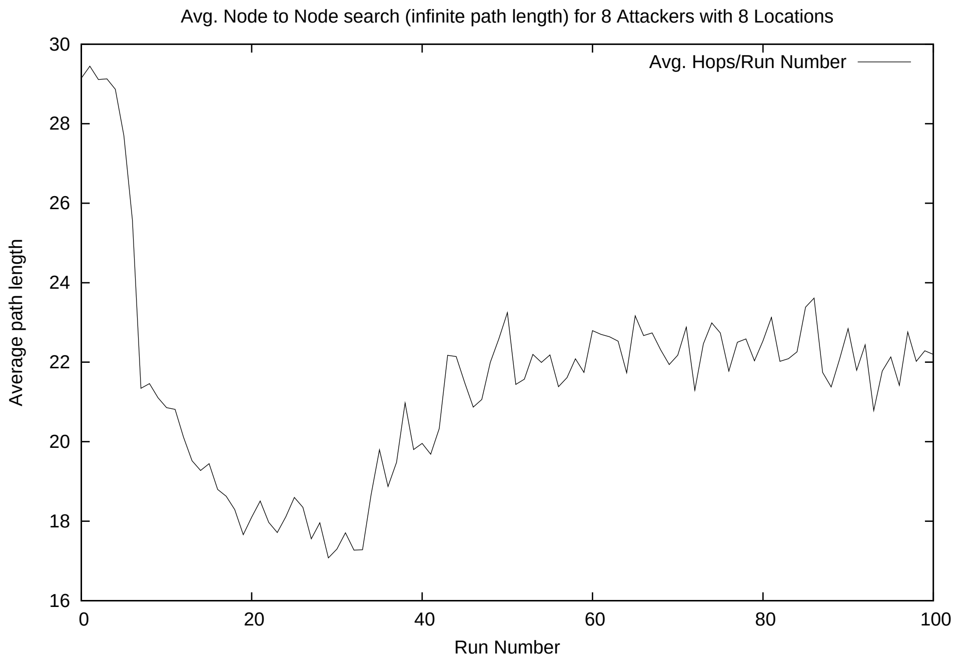


Figure 5.10. Graph showing average path length of completed requests when routing is unbounded.

between the initiator and the destination node, and also a count of all the hops along all the paths that were traversed prior to the desired node being found. We utilize both of these measures because it is important to know how many actual nodes are traversed in a GET or PUT request, though if the cost of initial message to discover the destination is low, the best path length found could be desired. Figures 5.11 and 5.12 show the number of hops between source and destination and the cumulative hops traversed including dead-end paths, respectively. As described previously, routing down a particular path fails when the path length is 10. These graphs show the large discrepancy between cumulative hops traversed before a query is successful and counting only those along the successful path. Although the cumulative lengths decrease for the duration of the experiment (with a noticeable spike when the attack is started) the lowest number of hops is still high. This may be an important factor when considering the costs to the network routing these requests. Counting only the successful path lengths reveals another aspect of the Freenet 0.7 network; while the un-converged network has a low number of hops and convergence decreases the hops, our attack nearly removes all the gains of this convergence.

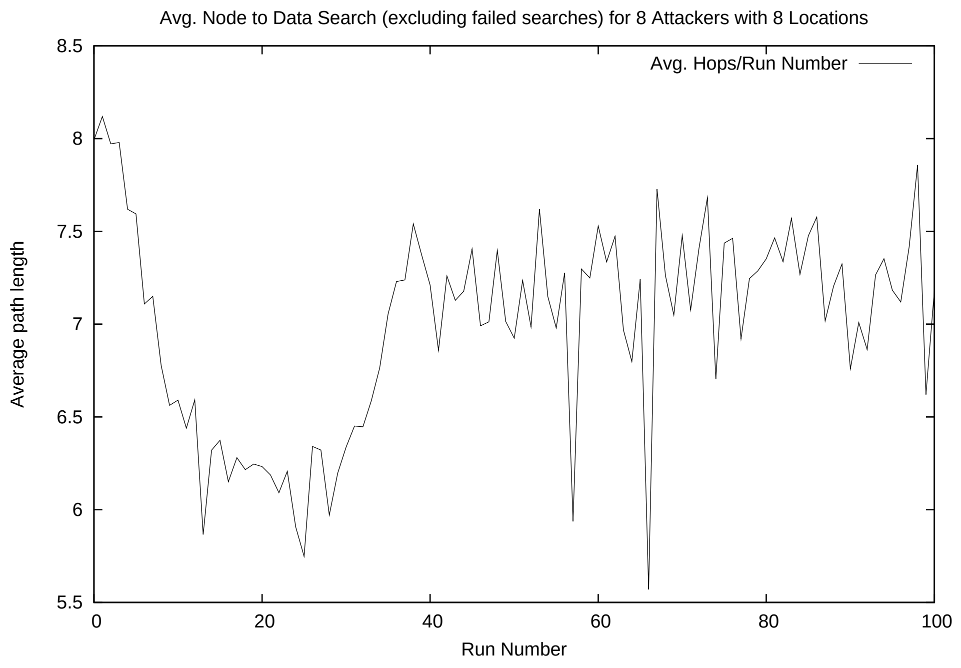


Figure 5.11. Graph showing average path length from source to destination of completed requests.

When searching for data in the network we know a priori whether or not a node is actually storing data (as we insert it at the appropriate node for each run). For efficiency we do not attempt to route these requests for missing data, but assume that such a message would traverse all nodes in the network before terminating. This is a reasonable assumption given that nodes have $O(\log n)$ neighbors and the network diameter is assumed to be $O(\log n)$, which is less than the maximum path length. This leads to our last measure, where we account for failed searches as taking n hops where n is the network size. The final graph in this section demonstrates truly the

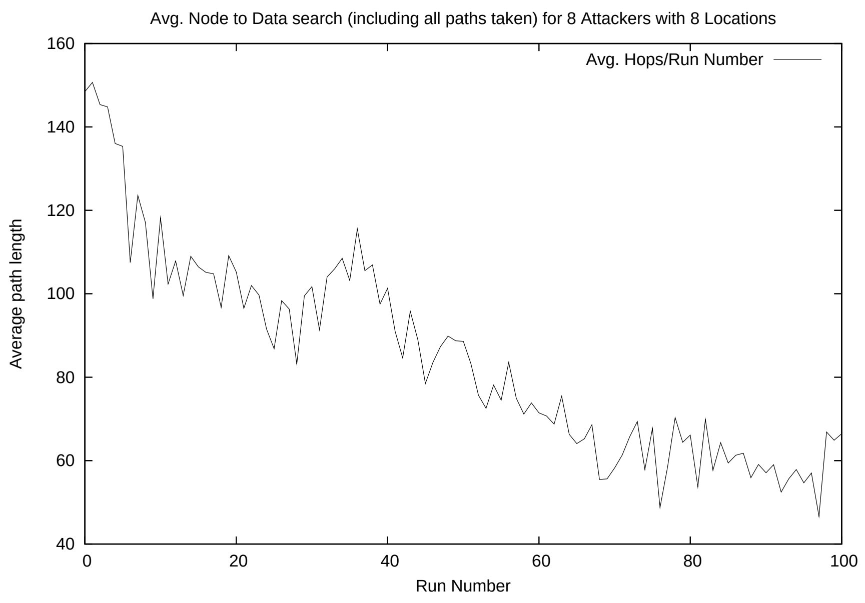


Figure 5.12. Graph showing average cumulative path length including all nodes traversed of completed requests.

efficacy of our attack on the Freenet 0.7 network in practice. Figure 5.13 shows very clearly that path lengths are increased significantly due to our attack for the simple reason that data is lost due to the high amount of clustering in the network. Note that there are failed requests even before the attack begins because routing is bounded.

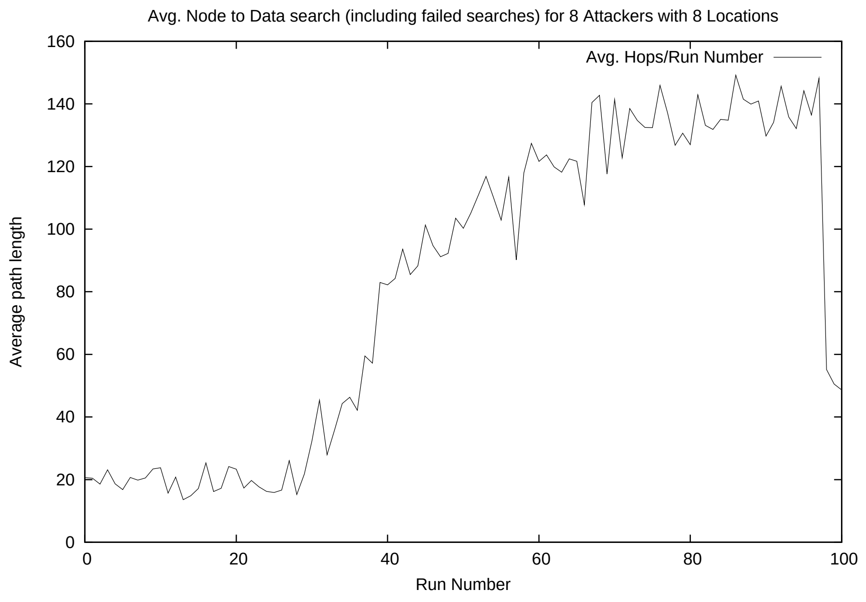


Figure 5.13. Graph showing average path length of requests when failed searches are assumed to traverse all nodes.

5.3 Availability of Content

Figures 5.14, 5.15 and 5.16 show the data loss in a simulated Freenet 0.7 network with 800 nodes and two, four and eight attackers respectively. The attackers attempt to use swapping in order to cluster the locations of nodes in the network around eight pre-determined values. The resulting clustering of many nodes around particular locations causes the remaining nodes to be responsible for disproportionately large areas in the key space. If this content assignment requires a particular node to store more content than the node has space available for, content is lost.

The attack is initiated after 75 iterations of ordinary network operation. After just 200 iterations the network has lost on average between 15% and 60% of its content, depending on the number of attackers. Note that in our model, an individual attacker is granted precisely the same resources as any ordinary user and if the attacker is deemed the node responsible for the data it will return it when queried. Obviously if the attacker were truly malicious, it could just drop the data, or refuse to answer queries that reach it, which would further reduce the storage capacity of the network. Our results do not show an attacker that takes this further step to hinder data being found. The figures show the average data loss (with standard deviations) over five runs of our test bed. For each run, the positions of the attackers were chosen randomly among the 800 nodes.

The second set of figures shows the data loss in our 400 node network simulations again with the same number of attackers as the previous three Figures. Figure 5.17 shows the results when two attackers are present, Figure 5.18 shows the results for four attackers, and Figure 5.19 shows the results for eight attackers. Again the data is averaged, over three runs. The duration of the trial has an effect on how well our swapping attack works; more time between tests gives our attackers more time to

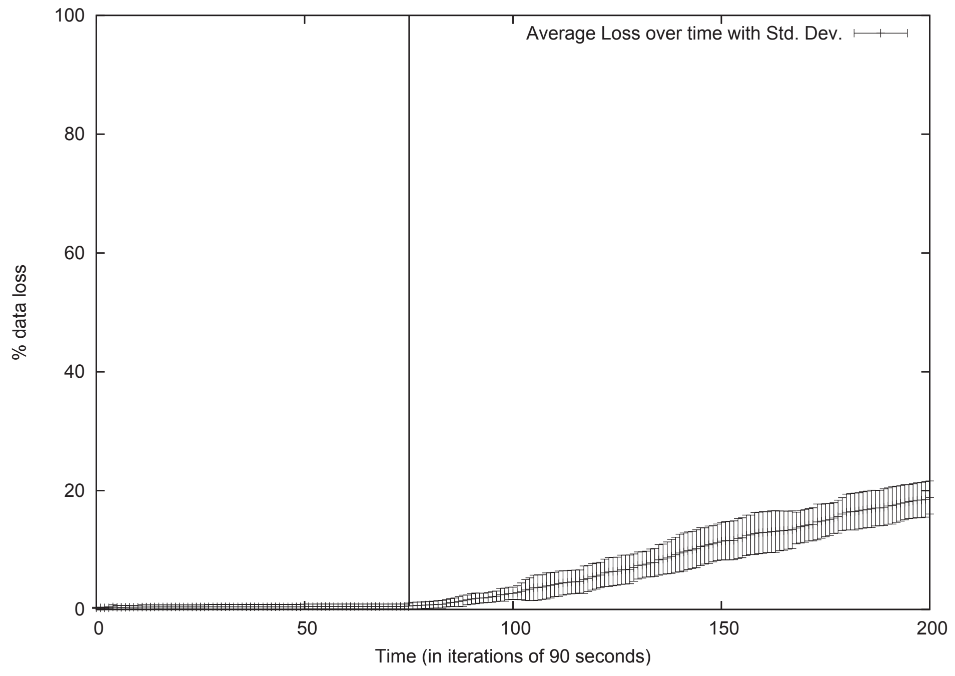


Figure 5.14. Graph showing average data loss over 5 runs with 800 total nodes and 2 attack nodes using 8 attacker chosen locations with the attack starting at iteration 75 **(horizontal line depicts attack start time).**

swap out malicious locations. Even so, we still see about 20% loss for two attackers, 30% for four attackers, and almost 60% for eight attackers. This is likely due to the size of the network, as with only 400 nodes the proportion of malicious nodes to normal nodes is higher.

Some criticisms may come of the fact that we seed the network with an evenly distributed quarter of the total storage capacity of the network. It may be suggested that this is too high of an initial amount of data for the distributed system to handle. This is really a moot point because changing the initial amount of data in the network only increases the time that it takes for an attacker to reduce the total storage capacity. The locations that malicious nodes propagate are so close together that once clusters of enough nodes are formed data *is* irrevocably lost. Given enough time

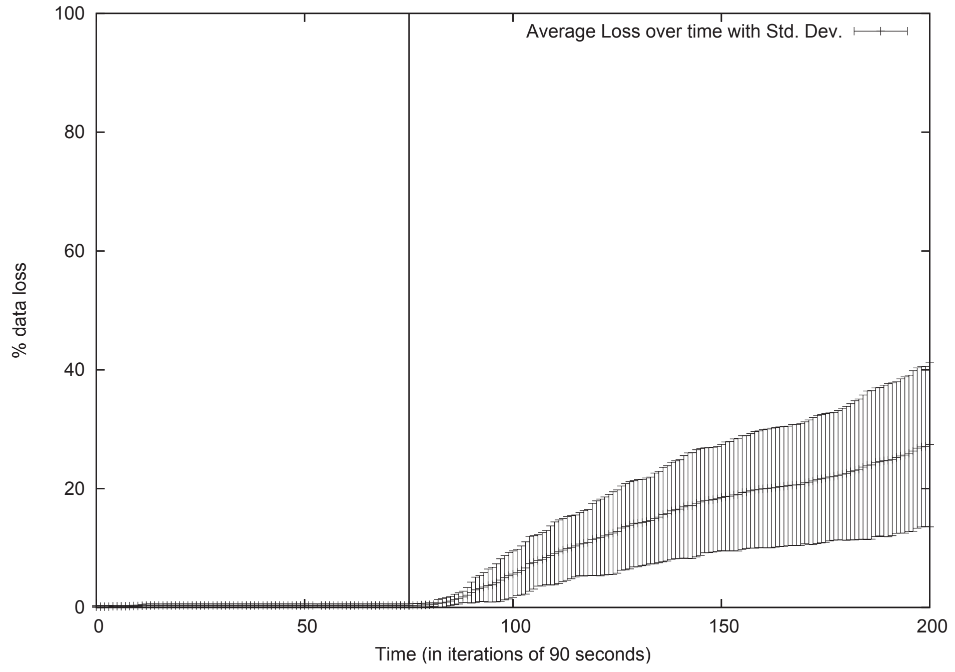


Figure 5.15. Graph showing average data loss over 5 runs with 800 total nodes and 4 attack nodes using 8 attacker chosen locations with the attack starting at iteration 75 **(horizontal line depicts attack start time).**

malicious nodes would cause the just as much data to be lost, but for our experiments to be short lived we chose what might be considered a high initial amount of data. As discussed in Section 5.2 the length the paths needed in order to find data in the network also plays an important part; because if the path to the data is too long the query will still fail even if the data remains in the network.

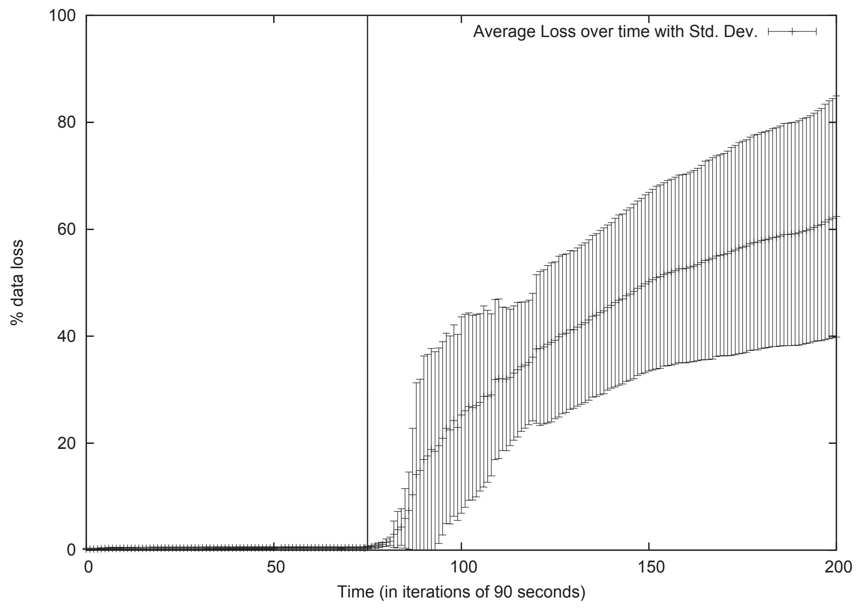


Figure 5.16. Graph showing average data loss over 5 runs with 800 total nodes and 8 attack nodes using 8 attacker chosen locations with the attack starting at iteration 75 (horizontal line depicts attack start time).

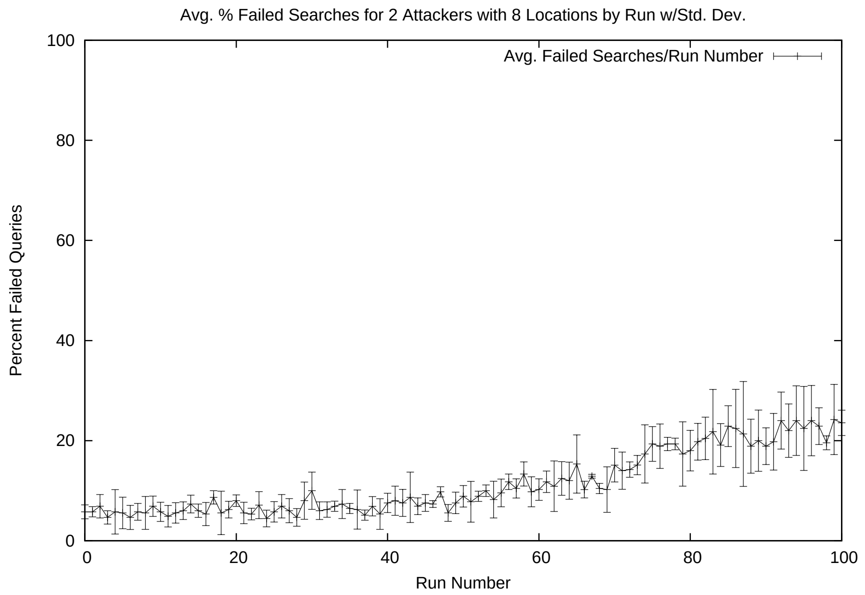


Figure 5.17. Graph showing average data loss in a 400 node network with 2 attack nodes using 8 attacker chosen locations with the attack starting at iteration 25.

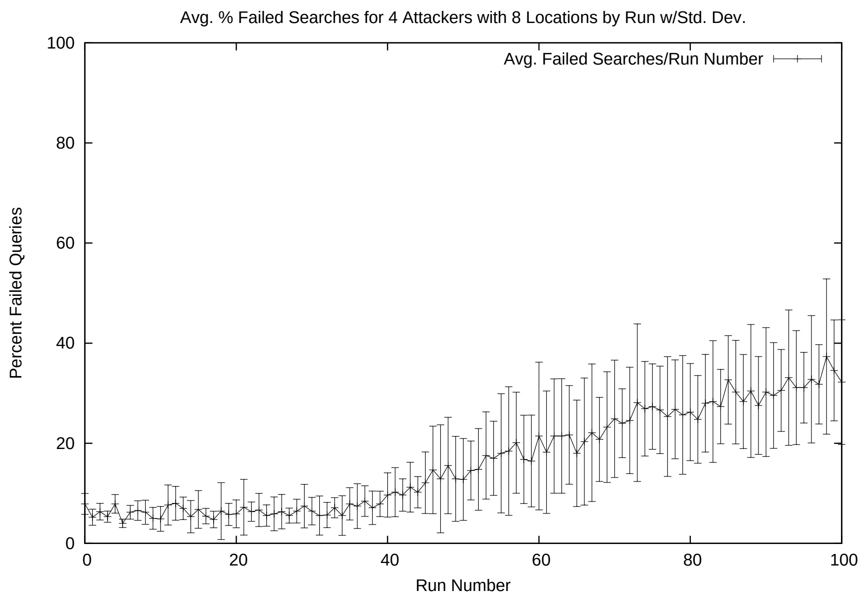


Figure 5.18. Graph showing average data loss in a 400 node network with 4 attack nodes using 8 attacker chosen locations with the attack starting at iteration 25.

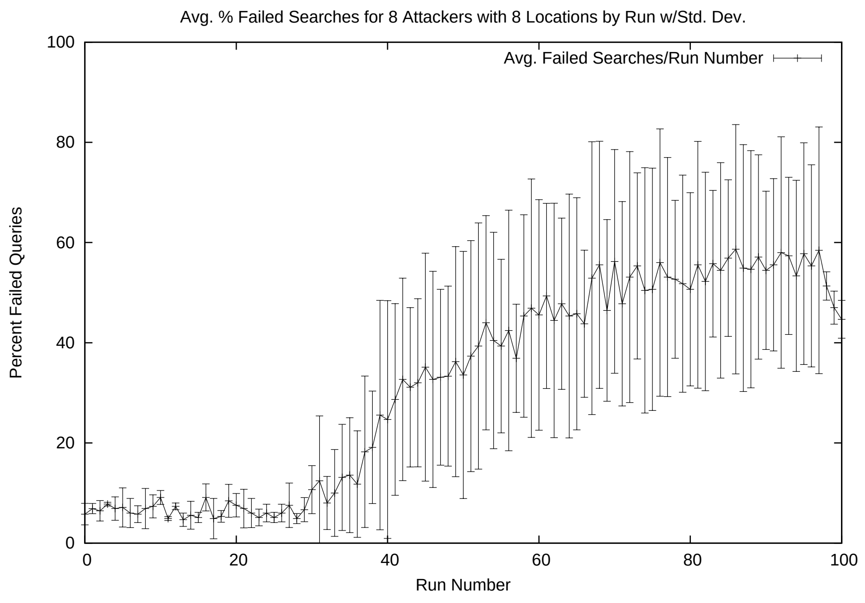


Figure 5.19. Graph showing average data loss in a 400 node network with 8 attack nodes using 8 attacker chosen locations with the attack starting at iteration 25.

5.4 Other Topologies

Although the Freenet 0.7 routing algorithm theoretically works best in small world networks with small world topologies, it is interesting to see how it performs in other topologies as well. We will demonstrate the differences between these topologies by comparing the metrics collected as described in Sections 5.2 for each of the topologies presented. As mentioned earlier, in addition to our "normal" small-world topology that we use for most of our experiments we introduce four modified topologies. Since we build our small-world topology by augmenting a 2d-grid, we thought it would be interesting to compare measurements with an un-augmented 2d-grid. In this case, each node in the network has exactly four connections corresponding to the nodes immediately above, below, right and left in the Cartestian system. We also wanted to see what the results looked like in the absence of the 2d-grid; meaning that we use the grid coordinates initially to find Cartesian distances between nodes (which is necessary for small-world construction see Section 3.1 for details) but we do not necessarily connect nodes into the 2d-grid. If two grid neighbor nodes are connected based on the randomized connection process they are not removed, so we are not forcing the absence of original grid connections, just not ensuring they are there.

Our other two topologies use the concept of "supernodes" [17], where certain nodes exist in the network that are much more highly connected (or universally connected) than "normal" nodes. We implemented supernodes very simply in our topology construction; when enabled, we selected probalistically 1 percent of the total nodes to be supernodes, which we augmented with an additional 20 percent of the total peers as direct connections. The 20 percent of nodes that were added as additional connections were also uniformly randomly selected from the set of all nodes, unlike the weighted random selection used for the small-world construction. We used these supernodes in two scenarios, with our normal small-world and with the simple 2d-grid topology.

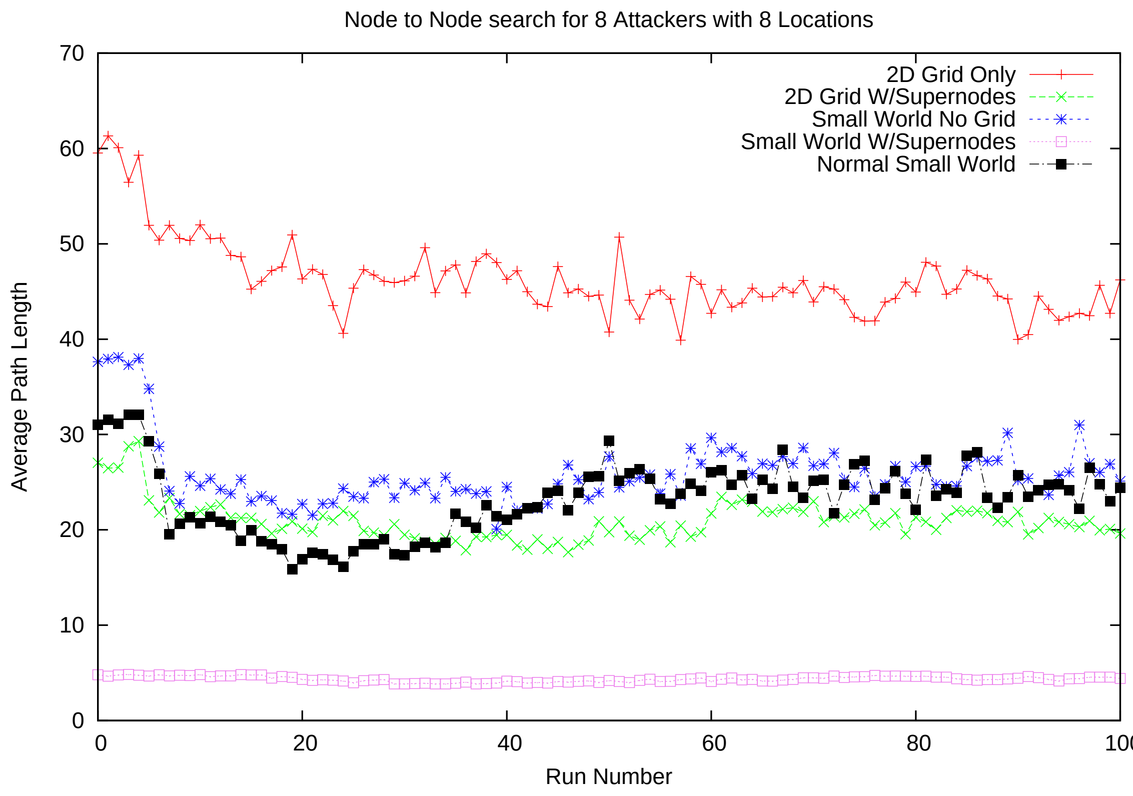


Figure 5.20. Average number of hops between source and destination for multiple topologies when there is no cap on path length.

The graphs in this section each have 5 lines, one for each topology and are labeled as such. To attempt to achieve a consistent comparison between each topology, we graph the results from tests with 400 nodes, 8 malicious nodes which are switched into attack mode at iteration 25, and the malicious nodes rotate through 8 locations over the course of the attack. The figures follow the same progression as those from Section 5.2. Figure 5.20 shows the average path length attained for successful queries when there is no bound on the path length in any one direction. This figure reveals two things that are very intuitive, the topology with the longest average path lengths is the simple 2d-grid and the one with the shortest average path lengths is the small-world network augmented with supernodes. This is exactly what one would expect because the 2d-grid is a very sparse graph and short paths do not exist between arbitrary nodes often. The small-world with supernode topology also should result in short path lengths because when a supernode is reached with a query it has at least a 1 in 5 chance of knowing the exact destination, and if not has a large pool to choose from for a closer node. The other three topologies are grouped together at about the same place on the graph, which is not unsurprising as they are very similar topologies as far as the number of total connections go. As in Section 5.2 we generally see average path lengths decreasing prior to the attack, and slightly increasing after the attack, with the exception of the supernode and 2d-grid only topologies.

Figures 5.21 and 5.22 again are best interpreted in conjunction. Due to the maximum path length for routing being 10, Figure 5.21 seems to show very short path lengths for all of the different topologies. Again we see that the best performing topology is clearly the small-world augmented with super-nodes, but at first blush they all show short paths. However, when the total cumulative hops are taken into account as in Figure 5.22 we see two more things. First we can see that the cumulative hops are considerably higher than the hops necessary once the data is found, and we also see that this cumulative number decreases as the network evolves over time. Again, only successful queries are counted in this figure, so the whole story is still not revealed.

Figure 5.23 probably provides the best argument for the properties inherent in small-world graphs based on a 2d-grid topology. In this graph the two small-world graphs and the 2d-grid augmented with supernodes are shown to be the least effected by our attack. The 2d-grid with supernodes is relatively the same across the board, which means that convergence due to the swapping algorithm and our attack have

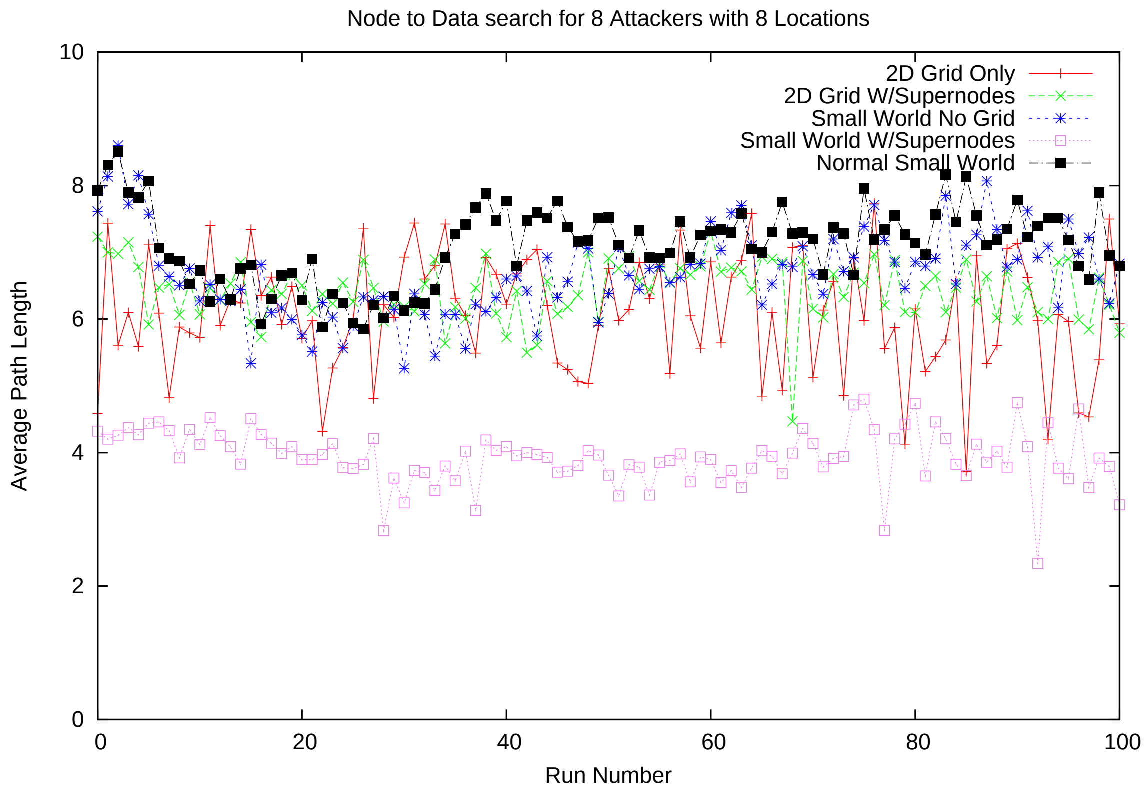


Figure 5.21. Average number of hops between source and destination for multiple topologies.

little effect on routing. The small-world with supernodes graph starts out lower than the normal small-world, but is more wild once the attack begins. The normal small-world topology performs the best in this instance, although the proportion of poisoned nodes is still high.

The final figure in this section shows the somewhat results of the number of poisoned nodes in the network as the attack progresses. This data, shown in Figure 5.24 is somewhat what one would expect. The normal small-world topology and the small-world with no grid show the biggest impact of our attack, with ¿50% of nodes poisoned by the time the attack ends. The 2d-grid only also shows a high number of poisoned

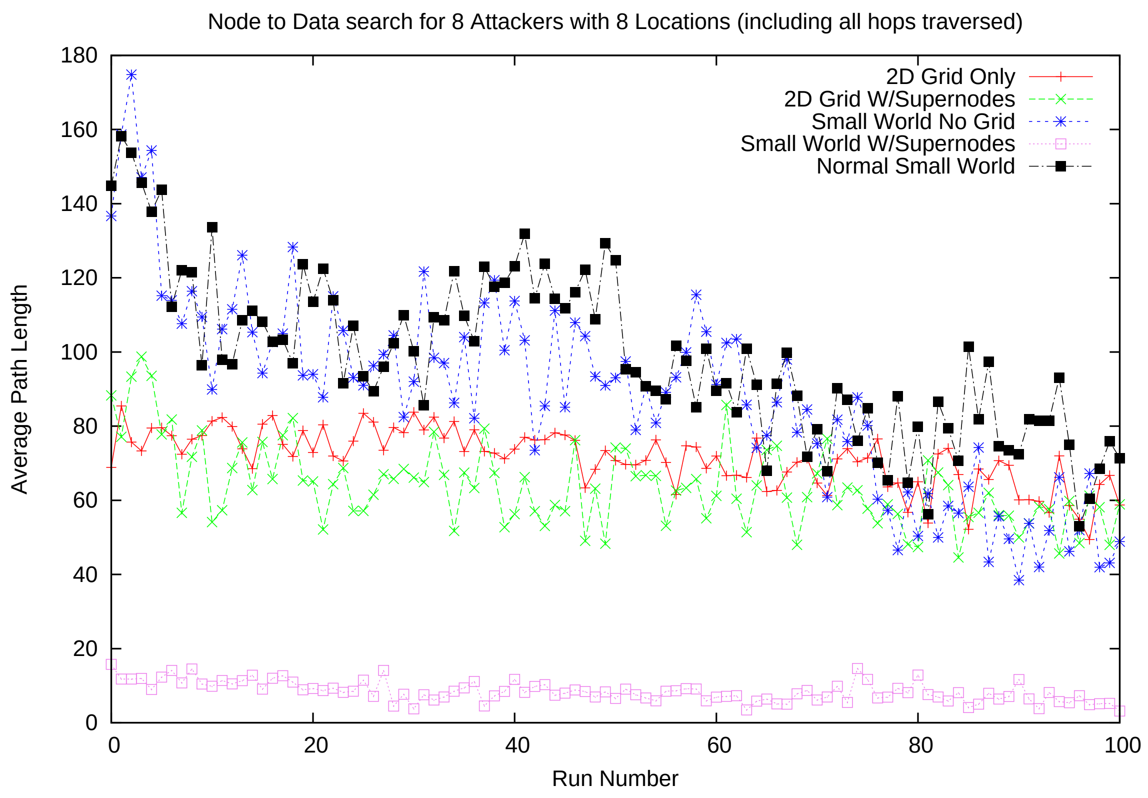


Figure 5.22. Average number of cumulative hops necessary for a query to succeed in multiple topologies.

nodes, presumably because attack locations spread out faster from the malicious nodes because each node has only 4 total connections, so the likelihood an attacker chosen location is close to the node's location goes up. The 2d-grid with supernodes shows the least number of poisoned locations, though still close to $\frac{1}{4}$ of all locations have been changed. The most surprising result is that of the low number poisoned for the small-world with supernode topology. However, the growth remains steady which is the only real requirement for us to prove that our attack works.

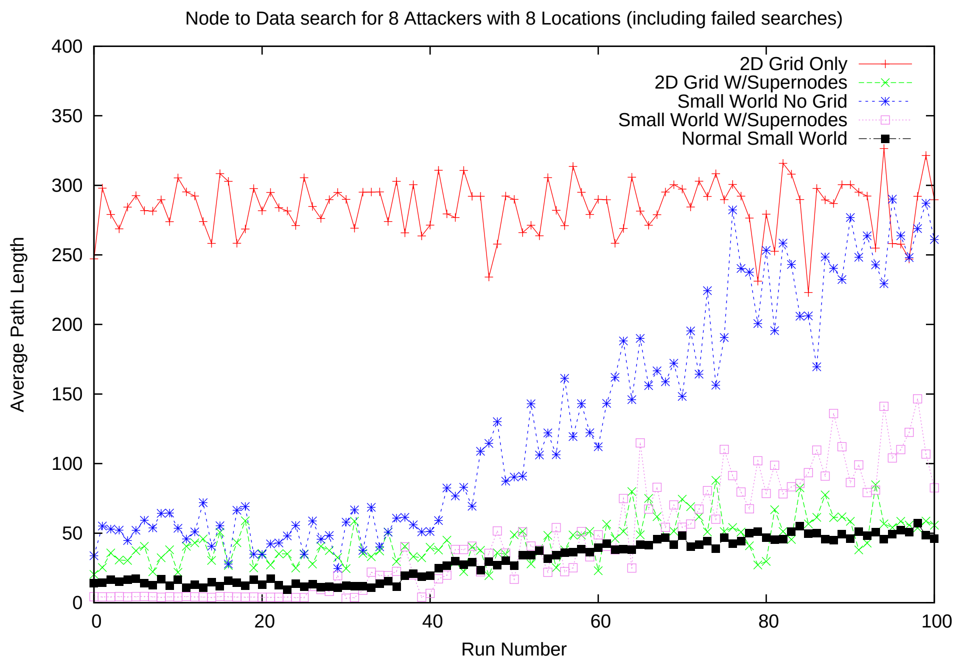


Figure 5.23. Average routing hops for queries in multiple topologies when failed searches are assumed to reach all nodes.

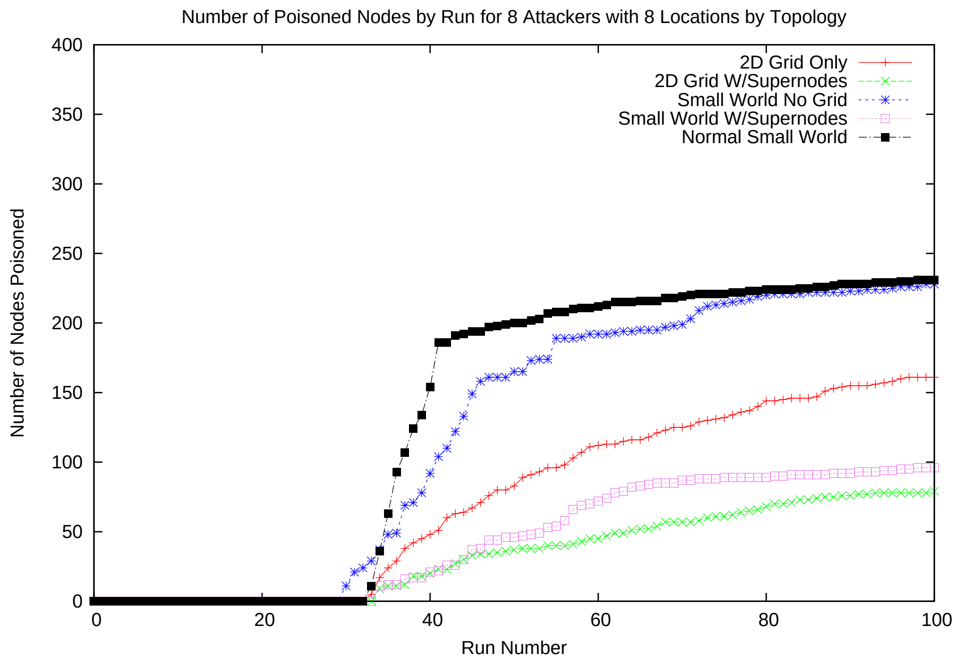


Figure 5.24. Number of poisoned nodes over time for multiple topologies.

5.5 Simulation of Churn

Figures 5.25, 5.26, 5.27,, 5.28, 5.29 and 5.30 show the results of a simulation of join-leave churn on the distribution of node locations in the Freenet 0.7 network. The total network size used for this simulation was 600 nodes, out of which 500 are stable. In each round, ten of the remaining hundred nodes drop out of the network and join as fresh nodes with a new random location and randomly chosen connections. During each round, the network performs the swap protocol corresponding to about 400ms of a real Freenet 0.7 network. The experiment was done with various different topologies with similar results. The figures show the results for a topology where the 500 stable nodes are randomly chosen from a small world network, that is, they are not better connected than any of nodes experiencing churn. In all of our simulations, the locations rapidly converge towards a small set of all the possible locations.

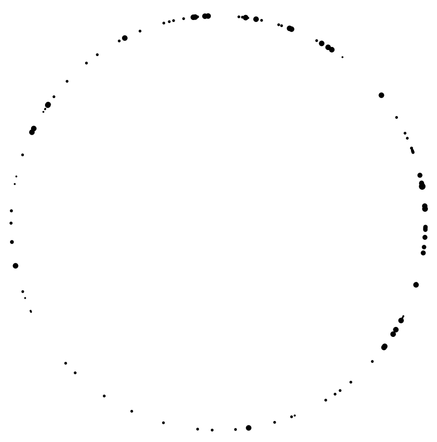


Figure 5.25. Initial (random) node locations before simulation of join-leave churn.

These results are particularly interesting because they show that the kind of location clustering which produced by our attack will also happen in an un-attacked network with a stable core of peers and churn. This is actually a very likely situation

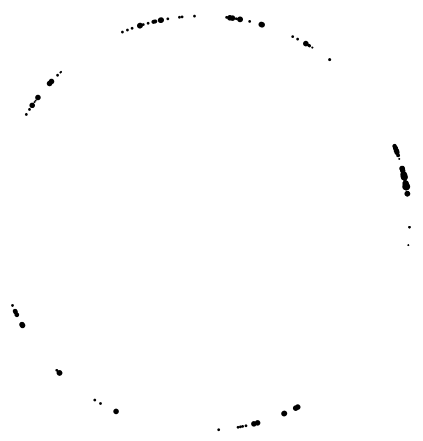


Figure 5.26. Distribution of the node locations of the stable core of 500 nodes after 100 rounds of churn.

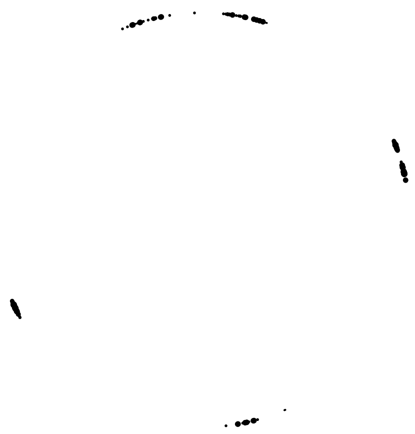


Figure 5.27. Distribution of the node locations of the stable core of 500 nodes after 2000 rounds of churn.

for any peer-to-peer network, as there are typically peers of developers/enthusiasts which are likely to be long lived. However a larger number of peers will join and leave as the network is tried out and then left due to a mismatch between what the

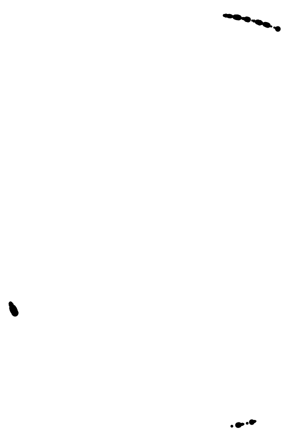


Figure 5.28. Distribution of the node locations of the stable core of 500 nodes after 10000 rounds of churn.

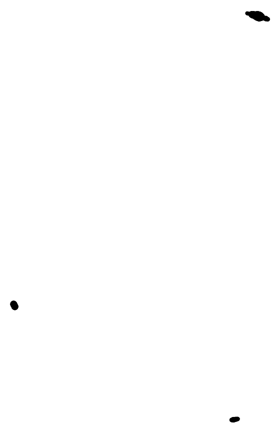


Figure 5.29. Distribution of the node locations of the stable core of 500 nodes after 50000 rounds of churn.

user wants and what the network provides. As we have shown in our Freenet 0.7 testbed, this kind of clustering is not good for the network, as it puts severe strain on those peers at the edges of clusters and generally unevenly distributes the processing

Figure 5.30. Distributions of the node locations of the stable core of 500 nodes after 100,000 rounds of churn.

and storage load of the entire network. In a distributed file system application like Freenet 0.7, uneven distribution is the the opposite of what the network design was created to achieve in the first place.

The negative impact of churn may be lessened by swapping locations only with long lived peers. Recent measurement studies in peer-to-peer networks have shown a power-law distribution of the uptime of peers; a large percentage of peers have a short uptime [57]. By adjusting the probability of location swapping to be proportional to the uptime of both peers, the network may be able to slow down the clustering of the locations of long-lived peers due to join-leave churn. This is only a conjecture however, we did not do experiments that show this to be a plausible way to reduce the clustering caused by churn but it is an interesting idea.

Chapter 6

Discussion

Various techniques have been proposed that may be able to limit the impact of the attack described in this thesis, including changing the swapping policy, malicious node detection, and secure multiparty computation. While some of these strategies can reduce the impact of the attack, we do not believe that adopting any of the suggested measures would address the attack in a satisfactory manner. We will now discuss these measures and their limitations insofar as we have identified them.

One possibility for reducing the effect of the attack proposed in this paper is to increase the amount of time between attempts to swap, or to have each node in the network periodically reset its location to a random value. The idea is that the malicious node locations would spread more slowly and eventually be discarded. However, while this would limit the impact of the attack, this defense also slows and limits the progress of the network converging to the most fortuitous topology. Specifically, the amount of time used between swaps must be chosen very carefully. In order to accurately decide on the swap frequency the number of nodes in the network (and the number of "poisoned" nodes, see below) need to be estimated. Both numbers are difficult to accurately attain. Also, as shown in Chapter 5, the malicious nodes are able to spread the “bad” locations chosen very quickly; within a few rounds of swapping a large proportion of the network has been poisoned. This implies that even frequent resets and increased swap frequency will do little to stop malicious nodes to any worthwhile degree. Another real problem with resetting nodes locations periodically is that in Freenet 0.7, data is assumed to be stored (and is actually stored) at the closest node to the data in the network. This means that when a node resets its location, the data that it is responsible for will likely no longer be at the closest node in the network. This would mean that either the data would be unable to be found, or measures to resend the data out to the correct nearest node would have to be implemented. Of course for data to stay in a network indefinitely it needs to be refreshed periodically, but if all nodes reset their locations in a short interval the network could be flooded with data being routed because its former nearest node is no longer very near!

Another possible method attempts to detect a malicious node based on knowing the size of the network. If a Freenet 0.7 node were able to accurately produce a close estimation of the size of the network, it could detect if an attacker was swapping locations that are significantly closer than what would be statistically likely based on a random distribution of locations. The problem with this approach is that in an open F2F network it is difficult to reliably estimate the network’s size. It is not easy for any single node to estimate the size of the network because the only information it knows is how many peers it has (and the peers of its neighbors). But this knowledge may be completely unrepresentative of the rest of the network. In fact, since the network is a “darknet” it would be very likely that nodes have largely varying numbers of peers. Any estimate based on such information would likely be wrong, or at least vary widely between peers in the network. One way to estimate the number of nodes in the network would be to take the number of peers and fit them in a distribution based on closeness. This would imply that if a node has many neighbors with locations close to its own, the assumption might be made that the network is very large. This could be completely wrong, especially when the clustering effect shown previously in a network with churn is at work! These reason make malicious node detection and altering the swapping frequency rife with problems.

If there were a way for a node which purported to have a certain number of friends to prove that all those friends existed, nodes could be more confident about swapping. The Freenet developers suggested using a secure multiparty computation as a way for a node to prove that it has n connections. The idea would be for the swapping peers to exchange the the results of a computation that could only be performed by their respective neighbors. But because nodes can only directly communicate with their peers (F2F), any such computation could easily be faked given appropriate computational resources. Of course, if a node could directly communicate with another node's neighbors, then the topology could be discerned. However, in that case the protocol no longer works for restricted-route networks. This method would also work provided that there were some trusted third party which gives out public/private key pairs to each node, but this is an unlikely step for an open peer-to-peer network to take for two reasons. First, peers joining a network have no reason to trust any third party, and second the third party would then know all of the members of the P2P network, which would defeat the idea of Freenet 0.7 being a "darknet" where only a node's friends know if that nodes participation in the network.

Chapter 7

Conclusion

The new Freenet 0.7 routing algorithm is unable to maintain scalability in networks where even weak adversaries are able to participate. The performance of the routing algorithm also degenerates over time (even without active adversaries) if the network experiences sufficient churn. Churn is unfortunately a very real problem in modern peer-to-peer networks and is therefore a serious limitation of the Freenet 0.7 darknet network. The approach chosen (by Freenet developers [24]) to address both of these problems is to periodically reset the locations of peers. Even this simple seeming solution is fraught with problems; while it limits the deterioration of the routes through adversaries and churn, such resets also sacrifice the potential convergence towards highly efficient routes. In order for efficient routes to remain possible the reset period would need to be chosen properly and to do so would require global knowledge of the network, which is very much against the idea of a darknet. While Freenet 0.7 was a good attempt at providing a solution to routing in restricted route networks, there are some serious problems that remain difficult if not impossible to solve. Secure and efficient routing in restricted route networks remains an open problem.

This thesis presents a substantial research topic that demonstrates a clear understanding of routing problems in P2P networks as they exist today. We not only discovered a possible attack vector, but we implemented a real world attack and demonstrated its feasibility and effectiveness on the Freenet 0.7 network. We also explained a phenomenon which was observed in practice by Freenet users but remained unresolved until the presentation of our topic to the Freenet developers. Through experiments performed on a Freenet 0.7 testbed which are comparable in size and operation to the actual network we have shown the effects of malicious nodes performing a modified swapping algorithm, and the clustering of nodes, effect on routing and significant data loss that are a direct result. By running simulations which emulated churn in a network we explained how clusters can form naturally in the Freenet 0.7 network over time, causing similar results as shown in our experiments. This research took a significant effort (spanning more than 15 months) to achieve our results, and has been recognized as such by the Freenet developers [24], ACSAC [19] and Defcon [18].

Bibliography

[1] E. Adar and B. A. Huberman. Free riding on gnutella. Technical report, Xerox Parc, Aug. 2000.

[2] R. Anderson. The Eternity Service. In *Proceedings of Pragocrypt 1996*, 1996.

[3] S. M. Bellovin. Security aspects of napster and gnutella, 2000.

[4] M. Casado and M. J. Freedman. Illuminating the shadows: Opportunistic network and web measurment. http://illuminati.coralcdn.org/stats/, December 2006.

[5] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *In Proceedings of the 5th Usenix Symposium on Operating Systems Design and Implementation*, 2002.

[6] I. Clarke. The freenet project. http://freenetproject.org/, 2007.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In H. Federrath, editor, *Designing privacy enhancing technologies: International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, July 25–26, 2000: proceedings*, volume 2009 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., 2001.

[8] B. Cohen. Incentives build robustness in bittorrent. Technical report, bittorrent.org, 2003.

[9] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *The ACM SIGCOMM'02 Conference*, August 2002.

[10] P. Criscuolo. Distributed denial of service - trin00, tribe flood network, tribe flood network 2000, and stacheldraht. Technical Report CIAC-2319, Department of Energy - CIAC (Computer Incident Advisory Capability), Feb. 2000.

[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 (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[12] G. Danezis, C. Lesniewski-laas, M. F. Kaashoek, and R. Anderson. Sybil-resistant dht routing. In *In ESORICS*, pages 305–318. Springer, 2005.

[13] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In H. Federrath, editor, *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*. Springer-Verlag, LNCS 2009, July 2000.

[14] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion

router, 2004.

[15] J. R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[16] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. Technical report, Microsoft Research, 2002.

[17] C. DSS. Gnutella protocol specification v0.4.

[18] N. S. E. et. al. Routing in the dark: Pitch black, August 2007.

[19] N. S. Evans, C. GauthierDickey, and C. Grothoff. Routing in the dark: Pitch black. In *ACSAC*, pages 305–314. IEEE Computer Society, 2007.

[20] M. T. Goodrich, M. J. Nelson, and J. Z. Sun. The rainbow skip graph: a fault-tolerant constant-degree distributed data structure. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 384–393, New York, NY, USA, 2006. ACM Press.

[21] J. H. Hartman, I. Murdock, and T. Spalink. The swarm scalable storage system. In *International Conference on Distributed Computing Systems*, pages 74–81, 1999.

[22] M. Herlihy and J. D. Tygar. How to make replicated data secure. In *CRYPTO*, pages 379–391, 1987.

[23] K. Horowitz and D. Malkhi. Estimating network size from local information. *Information Processing Letters*, 88:237–243, 2003.

[24] M. T. Ian Clarke and Others. The devl archives (freenet), August 2007.

[25] E. jin Goh. Secure indexes. In *In submission*, 2004.

[26] E. jin Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *in Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pages 131–145, 2003.

[27] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 29–42, Berkeley, CA, USA, 2003. USENIX Association.

[28] J. M. Kleinberg. Navigation in a small world. *Nature*, 406(6798):845–845, August 2000.

[29] J. M. Kleinberg. The small-world phenomenon: an algorithm perspective. In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 163–170, New York, NY, USA, 2000. ACM Press.

[30] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[31] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[32] R. Levien. Attack resistant trust metrics. Draft available at http://www.levien.com/thesis/compact.pdf, 2003.

[33] E. Mane, E. Mopuru, K. Mehra, E. Mane, E. Mopuru, K. Mehra, and J. Srivastava.

Network size estimation in a peer-to-peer network, 2005.

[34] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02, Cambridge*, pages 53–65, March 2002.

[35] D. Mazières. *Self-certifying file system.* PhD thesis, MIT, 2000.

[36] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, 2002.

[37] S. Milgram. The small-world problem. *Psychology Today*, pages 60–67, 1967.

[38] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 31–44, New York, NY, USA, 2002. ACM.

[39] G. Pandurangan. Building low-diameter p2p networks. In *FOCS '01: Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, page 492, Washington, DC, USA, 2001. IEEE Computer Society.

[40] G. Perng, M. K. Reiter, and C. Wang. Censorship resistance revisited. In J. Herrera-Joancomarti, editor, *Pre-Proceedings of the 7th International Workshop on Information Hiding*, pages 279–293, 2005.

[41] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

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

[43] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. 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.

[44] M. K. Reiter and A. D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[45] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *In Proc. of USENIX Technical Conf*, pages 10 – 10, 2004.

[46] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489 (Proposed Standard), Mar. 2003. Obsoleted by RFC 5389.

[47] J. Rosenberg, J. Weinberger, P. Matthews, and R. Mahy. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun), Oct. 2008.

[48] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science*, 2218:329 – 350, 2001.

[49] A. I. T. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[50] O. Sandberg. *Searching in a Small World.* PhD thesis, University of Gothenburg and

Chalmers Technical University, 2005.

[51] O. Sandberg. Distributed routing in small-world networks. In *ALENEX*, 2006.

[52] A. Shamir. How to share a secret. In *Communications of the ACM*, volume 22, pages 612–613. ACM, 1979.

[53] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[54] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[55] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663 (Informational), Aug. 1999.

[56] 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*, pages 149–160. ACM Press, 2001.

[57] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM Press.

[58] M. Waldman and D. Mazières. Tangler: A censorhip-resistant publishing system based on document entanglements. In *ACM Conference on Computer and Communications Security*, pages 126–135, 2001.

[59] D. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[60] B. Wilcox-O'Hearn. Experiences Deploying a Large-Scale Emergent Network. In *Peer-to-Peer Systems: First International Workshop, ITPTS 2002*, pages 104–110. Springer-Verlag Heidelberg, January 2002.

[61] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003. Special Issue on Service Overlay Networks, to appear.