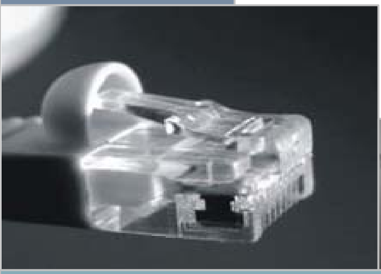
Network Architectures
and Services
NET 2011-08-1

Dissertation

Methods for Secure Decentralized Routing in Open Networks

Nathan S. Evans

Network Architectures and Services
Department of Computer Science
Technische Universität München



Technische Universität München
Lehrstuhl für Netzarchitekturen und Netzdienste

Methods for Secure Decentralized Routing in Open Networks

Nathan S. Evans

Vollständiger Abdruck der von der Fakultät für Informatik
der Technischen Universität München
zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Thomas Neumann

Prüfer der Dissertation:

1. Christian Grothoff, Ph.D (UCLA)
2. Professor Mikhail J. Atallah
3. Univ.-Prof. Dr. Thomas Huckle

Die Dissertation wurde am 22. Juni 2011 bei der Technischen Unversität München eingereicht und durch die Fakultät für Informatik am 10. August 2011 angenommen.

Cataloging-in-Publication Data
Nathan S. Evans
Methods for Secure Decentralized Routing in Open Networks
Dissertation, August 2011
Network Architectures and Services, Department of Computer Science
Technische Universität München

ISBN: 3-937201-26-2
ISSN: 1868-2634 (print)
ISSN: 1868-2642 (electronic)
Network Architectures und Services NET 2011-08-1
Series Editor: Georg Carle, Technische Universität München, Germany

ABSTRACT

The contribution of this thesis is the study and improvement of secure, decentralized, robust routing algorithms for open networks including ad-hoc networks and peer-to-peer (P2P) overlay networks. The main goals for our secure routing algorithm are openness, efficiency, scalability and resilience to various types of attacks. Common P2P routing algorithms trade-off decentralization for security; for instance by choosing whether or not to require a centralized authority to allow peers to join the network. Other algorithms trade scalability for security, for example employing random search or flooding to prevent certain types of attacks. Our design attempts to meet our security goals in an open system, while limiting the performance penalties incurred.

The first step we took towards designing our routing algorithm was an analysis of the routing algorithm in Freenet. This algorithm is relevant because it achieves efficient (order $O(\log n)$) routing in realistic network topologies in a fully decentralized open network. However, we demonstrate why their algorithm is not secure, as malicious participants are able to severely disrupt the operation of the network. The main difficulty with the Freenet routing algorithm is that for performance it relies on information received from untrusted peers. We also detail a range of proposed solutions, none of which we found to fully fix the problem.

A related problem for efficient routing in sparsely connected networks is the difficulty in sufficiently populating routing tables. One way to improve connectivity in P2P overlay networks is by utilizing modern NAT traversal techniques. We employ a number of standard NAT traversal techniques in our approach, and also developed and experimented with a novel method for NAT traversal based on ICMP and UDP hole punching. Unlike other NAT traversal techniques ours does not require a trusted third party.

Another technique we use in our implementation to help address the connectivity problem in sparse networks is the use of distance vector routing in a small local neighborhood. The distance vector variant used in our system employs onion routing to secure the resulting indirect connections. Materially to this design, we discovered a serious vulnerability in the Tor protocol which allowed us to use a DoS attack to reduce the anonymity of the users of this extant anonymizing P2P network. This vulnerability is based on allowing paths of unrestricted length for onion routes through the network. Analyzing Tor and implementing this attack gave us valuable knowledge which helped when designing the distance vector routing protocol for our system.

Finally, we present the design of our new secure randomized routing algorithm that does not suffer from the various problems we discovered in previous designs. Goals for the algorithm include providing efficiency and robustness in the presence of malicious participants for an open, fully decentralized network without trusted authorities. We provide a mathematical analysis of the algorithm itself and have created and deployed an implementation of this algorithm in GNUnet. In this thesis we also provide a detailed overview of a distributed emulation framework capable of running a large number of nodes using our full code base as well as some of the challenges encountered in creating and using such a testing framework. We present extensive experimental results showing that our routing algorithm outperforms the dominant DHT design in target topologies, and performs comparably in other scenarios.

Acknowledgments

Chapter 2 is based on work previously presented at ACSAC 2007 [59], and was extended into the author's master thesis [57]. The work presented in Chapter 3 was published at P2P 2010 [122]. Chapter 4 is strongly based on our paper published at Usenix Security 2009 [58]. Chapter 6 is a longer version of a workshop publication published at CSET'11 [60]. Chapter 7 contains material which was, at the time of this writing, accepted for publication at NSS 2011 [56].

This research was supported in part by the NLnet Foundation from the Netherlands (`http://nlnet.nl/`), under NSF Grant No. 0416969 and by the Deutsche Forschungsgemeinschaft (DFG) under ENP GR 3688/1-1.

I would like to thank my advisor, Christian Grothoff, for his guidance and patience throughout the course of my graduate career. Additionally, I would like to thank other co-authors and contributors to my research efforts. Specifically, Chris GauthierDickey was instrumental in the research which led to [59]. Roger Dingledine provided key insight into Tor and asked excellent questions which made the work presented in [58] that much more applicable to the real world. Samy Kamkar came up with the initial technique presented in [122], and Andreas Müller provided the testbed upon which our implementation was evaluated on real world NATed networks. Without the ideas, critical evaluation and writing skills of these individuals none of this research would have been possible.

Special thanks are due to Ramki Thurimella and Krista Grothoff for their "behind the scenes" efforts which have contributed both to the quality (and understandability) of the research presented in this thesis, and to my personal growth throughout the course of my graduate career. In particular, Ramki has often provided me with valuable advice on papers, presentations and more. Krista has tirelessly put up with bad writing and edited countless documents without remuneration.

I am unable to thank Katie Haus enough for her unending support and encouragement in all aspects of my life throughout the course of this research. She has shown patience, creativity and resolve; at times providing solace, acting as muse, and even doing the dirty work of editing documents. Many of the figures throughout this thesis have her touch, and she is also very good at math!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Routing algorithms are at the heart of any large-scale networking application including P2P file-sharing [32, 49, 148, 173], P2P conferencing [52, 186] and interactive multi-player gaming [48]. Of course, any Internet application utilizing TCP or UDP uses IP routing. Despite the ubiquity and necessity of routing algorithms, most have serious security problems. These algorithms enable participating malicious operators to potentially disrupt network operations. Most operators are trustworthy most of the time; however, as we show in Chapters 2 and 4, a few malicious participants often have the ability to circumvent fundamental security goals of the respective network.

A common task for distributed systems is data storage. Distributed hash tables (DHTs) are a type of distributed data structure that provide operations to store and retrieve key/value pairs. Existing DHTs typically accomplish routing in $O(\log n)$ steps under the assumption that any node in the network is able to directly communicate with any other node in the network [112,153,173,187]. Security designs for existing DHTs [144,154,167] typically assume trusted authorities authenticating the participants, thereby limiting the participation of malicious peers.

This thesis covers the motivation, related work, design, implementation and evaluation of a secure DHT that avoids the need for participants or operators to be authenticated by a trusted authority. We call the resulting network *open* since it allows new users to join at any time without pre-conditions. Another major difference between our design and previous designs is that we do not assume universal connectivity between all peers in the network; we only require that short paths exist between all peers as is the case in Small-World networks [183]. We term networks which lack universal connectivity *restricted-route* networks, including networks restricted by network address translation (NAT), firewalls or physical restrictions. Small-World networks are a specific type of restricted-route network that fulfills the "short path" requirement; these are the topologies our design is primarily focused on.

The remainder of this chapter further describes and motivates the operating environment and goals for our system. In Chapter 2 we show that Sandberg's secure routing algorithm for restricted-route networks [158] is in fact, not secure. Specifically, participating malicious nodes can severely degrade the performance of the routing algorithm and the storage capacity of the network. Chapter 3 outlines the need for NAT traversal [151] techniques in any modern P2P network, and details our NAT punching technique which operates without the use of a trusted third party. NAT traversal is necessary for P2P networks to improve connectivity, especially when significant portions of Internet users are behind NAT. Most P2P networks assume universal connectivity, so more connections than are possible with NAT traversal is generally useful. Chapter 4 describes a design flaw discovered in and an attack on Tor [45]; which uses onion routing to anonymize peer's actions. This practical attack prompted us to avoid the design flaw in our implementation of a locally bounded distance vector protocol which includes features similar to Tor. Chapter 5 briefly describes the motivation and implementation of the distance vector protocol designed to provide additional connectivity in restricted-route networks. Like NAT traversal, increasing connectivity in restricted route networks is desirable so that higher level routing algorithms, such as our DHT routing algorithm, are able to route efficiently. In order to show that our routing algorithm performs as expected under varying conditions, we have created a fully distributed emulation framework which we use to perform large scale testing. This testing and profiling framework is described in Chapter 6. Chapter 7 outlines our design of a robust and simple routing algorithm and discusses why our algorithm improves upon extant designs. We provide a mathematical analysis of certain aspects of the routing algorithm and reflect on the effects of our design choices. Finally, Chapter 7 provides a wide range of experimental results obtained using our testing framework. These results showcase the performance of an implementation of our design under multiple topologies, network sizes and in the presence of malicious participants.

1.1 Network Environment

The Internet was intended to be a decentralized system of organizations and networks providing end-to-end connectivity between any two hosts using TCP/IP [26]. Addressing on the Internet is hierarchical, with Internet service providers (ISPs) assigning contiguous sub-ranges of numerical addresses to their customers. These hierarchical address spaces form Autonomous Systems (ASes) within which simple intra-domain prefix-based routing algorithms such RIP [78] and OSPF [121] executed by routers under the control of a single trusted entity are the norm.

There are many attacks on these low level routing algorithms [12–14, 77], such as forcing routing on a subnet to go through a specific malicious host or denying access to specific IP addresses. These problems are well understood and workarounds exist [23, 50, 83, 121]; however, they typically require network operations to be authorized by a trusted authority, and do not address scalability limitations of these routing algorithms.

Routing between ASes is typically controlled by the Border Gateway Protocol (BGP) [145]. BGP can be tricked by malicious or incompetent AS operators into routing information to the wrong AS or via inefficient paths. There are two recent infamous examples of exploiting BGP in this way. The first happened when Pakistan ISP's attempted to block YouTube [84] by redirecting Pakistani users to a null interface. A misconfiguration updated the routes for most Internet users, thus globally blocking most access to YouTube [31]. This was a mistake, but more intelligent intentional efforts using the same family of techniques allows man-in-the-middle attacks where inter-domain traffic can be undetectably viewed by a third party [88]. It remains a mystery whether the second incident was a mistake or intentional. Regardless, in April, 2010 a Chinese AS operator claimed roughly 15% of all Internet routes for around 20 minutes [19]. The possibility of combining this type of attack with the insecurity inherent in the extant public key infrastructure (PKI, Section 1.1.3) widely used to secure Internet communication calls into question how "secure" these communications truly are.

In addition to BGP, there are a number of protocols and systems in place on the Internet that form the basis of most IP communication. It is important to understand what constraints we are under when designing a protocol for the Internet, so we describe these realities and some of the problems that they can lead to in the remainder of this section.

1.1.1 Network Address Translation (NAT)

NAT was introduced to extend the lifetime of IPv4 (the main addressing scheme used on the Internet) because of address space exhaustion. NAT [54, 168] has made the assumption of universal connectivity between devices on the Internet unrealistic. NAT hides multiple addresses on an internal network behind one globally routable address. Specifically, this means that computers behind the NAT device can initiate communication with globally routable addresses, but globally routable addresses cannot generally initiate communications to devices on the internal network. Some estimates put the number of Internet systems behind NAT at up to 70% [27].

NAT clearly creates problems for applications that rely on universal connectivity. This has forced designers of network applications to account for limitations imposed by NAT. Common methods for dealing with NAT include NAT punching, STUN and UPnP. We have developed a new autonomous NAT traversal technique, described in Chapter 3. None of these techniques are universally successful at providing connectivity to peers behind NAT. Some networks are restricted for other reasons, such as ad-hoc wireless networks where physical distance between peers makes certain connections impossible. Thus, applications designed to work for NATed users cope with problems resembling those commonly found in ad-hoc networks, including wireless sensor networks. As a result, routing algorithms that work well for restricted-route networks will be applicable to a broad range of applications including Internet routing with NAT and ad-hoc networks.

1.1.2 Domain Name System (DNS)

The DNS [118] is arguably the most important application on the Internet. DNS enables canonical names to be resolved to IP addresses. DNS servers are responsible for storing the name to IP address mapping so that user applications can discover this information. DNS servers are arranged in a tree hierarchy with so-called *root* servers at the top. DNS servers at each subsequent level down store more specific information. In this way, the responsibility for storing mappings is distributed over the tree. When a user application needs to discover the IP address related with a canonical name the tree is traversed (first up then down) until the query is resolved. DNS name servers also cache results as an optimization, which allows known results to be returned without fully traversing the tree.

There have been major problems with DNS [11,155,160,172]; due to the treelike structure and caching. When a DNS name is "hijacked" or a cache "poisoned", clients retrieve an incorrect mapping between domain name and IP address, causing them to trust or route to the wrong IP address. Domain name system security extensions (DNSSEC) are a method currently being deployed to solve DNS security issues. DNSSEC makes each DNS record verifiable; allowing each client to check the name to IP address mapping. This removes the ability of an attacker to perform a man-in-the-middle attack by returning his IP address instead of the correct one and then monitoring the client/server connection. However, the true problem with DNSSEC is inherent in the design; a hierarchy of trust, where malicious behavior or benevolent incompetence at the root of the tree is disastrous.

In DNS, the canonical name can be thought of as a key being searched for, and the IP address is the data returned. Thus, DNS could be implemented over a secure DHT, which provides the same distributed key mapping. In fact, designs for such systems already exist [134], but have not seen widespread use. Using such a DNS over DHT solution would reduce the reliance of DNS on root servers. This could conceivably reduce some of the aforementioned problems with DNS.

1.1.3 Public Key Infrastructure

The most widely used method for providing secure communication on the Internet today is based on the secure sockets layer/transport layer security (SSL/TLS) [41], which is a protocol for encrypting communication between a client and server. This is based on a hierarchical public key infrastructure, where a set of public keys for certain organizations are known and trust can be delegated through trust chains. These high level, or "root" certificates are also known as the trust anchors of the system. They provide the basis from which the rest of the system derives its security. SSL works as follows; a client has a list of "trusted" public root certificates, the server (and possibly also the client) has obtained a private certificate signed by one of the private root certificates (or via intermediate certificate authorities or CA's). Based on this verifiable signature the client trusts that the server is valid, and from these certificates an encrypted tunnel can be created between client and server.

This system is easily understood, and has worked rather well since its inception, but there are a number of glaring flaws. The first problem is serious; clients are forced to put trust in certificate authorities may or may not actually be trustworthy. Common browsers contain up to 124 [51] root certificates, and there are upwards of 650 CA's which are able of sign certificates. It is impossible for users to actually trust all of these, and any of them can sign certificates for *any* domain. As an example, if a "rogue" CA signed a `citibank.com` certificate for a malicious operator, that operator could perform man-in-the-middle attacks for `citibank.com` users. More troubling, there have been reports of lax practices by certificate issuers, including issuing certificates for incorrect domains and wild-card certificates [51]. Also, government operated CA's such as those run by the United States department of homeland security and the Chinese government raise the possibility of *silent* interception, monitoring, etc. of "secured" communications for users.

There have been multiple proposed fixes for SSL, including certificate revocation lists (CRLs) and the on-line certificate status protocol (OCSP). CRLs are lists of previously issued certificates that are no longer valid. CRLs are a good idea, though modern browsers do not use them. Issues cited by browser developers include the dynamic nature and large size of CRLs which make distribution and updates prohibitively costly. OCSP is a protocol allowing clients to query a server about the status of a certificate. The server checks a live CRL and issues a response regarding certificate validity. This shifts the CRL update issue from each client to a centralized server. While some browsers support OCSP, it is up to the CAs to provide a working implementation; thus, a failure to receive a response to an OCSP request is not treated as a revocation.

Perhaps the best example of the insecurity of SSL and the failure of CRLs and OCSP is an incident that occurred in March, 2011. At this time, a "hacker" infiltrated a CA's systems and issued signed certificates for nine high risk domains including `login.live.com`, `mail.google.com` and `login.skype.com`. Using these certificates the attacker could have performed undetectable man-in-the-middle attacks. The most worrisome details of this attack are not directly observed consequences but how the breach was handled. Specifically, the CA in question waited a number of days before informing browser developers, and over 9 days before publicly revealing the breach. More worrying still, it may have never become public knowledge but for the sharp watch kept on CRL and browser updates released. An independent researcher released a blog report [3] detailing signs of the breach which may have prompted the public admission days later. Also, because of the sad state of CRLs and OCSP, browser *source code* modification (explicitly blocking the compromised certificates) was required. Only users that updated their browser binaries were protected from these "revoked" certificates. Clearly, this is not a tenable way to provide good security to users in the Internet.

1.1.4 Trust Agility

Mentioned previously, DNSSEC is one fix for the man-in-the-middle attack possible on the public key infrastructure which employs SSL for domain security. The problem with DNSSEC is that it is based on essentially the same chain-of-trust relationships as SSL. Trust is anchored in operating system defined certificates, just like SSL (only SSL's are in the browser), and trust is still delegated to third parties including top level domains and registrars. While DNSSEC is an improvement over DNS, it remains a system where users must put their trust in others, who may not always have user security at the forefront of their interests. This problem has been stated as a lack of so called "trust agility" [109], where users are unable to selectively choose where they anchor their trust, and be able to alter that choice in the future. With an open P2P network, on the other hand, there are no trust anchors. Replacing DNS with an open P2P based DHT changes the problem of trust to one of having the correct data stored in the network.

1.1.5 Summary

In this section we have described some of the issues that are prevalent in the networking environment that makes up the Internet. Due to the common usage of these designs and since they are required to function for the millions of Internet users, change comes very slowly to fix their issues. Some of these problems stem from how routing works in the network underlay. Some of these issues also limit the design of new routing algorithms. However, new techniques for routing have been developed, mainly in the domain of peer-to-peer (P2P) networks. Routing is important in P2P communication between peers at the overlay level. We provide an introduction to P2P networks in the next section.

1.2 Peer-to-Peer (P2P) Networks

Distributed systems are those that mete out work or responsibility across a set of nodes in order to pool the resources provided by the collective participants of the system. A pure P2P system is a distributed system in which all participants are "equal". This generally means that responsibility is evenly distributed among the members, and that no single node is more powerful than any other. This type of design has some clear advantages over centralized system designs, such as lacking a single point of failure, distribution of bandwidth or computation across peers, etc. P2P systems are commonly scalable; as more nodes are added to the system they add to the total resources of the system. In a client/server system, adding clients puts more load only on servers and therefore does not scale without adding more servers (which increases costs for the server operator). P2P systems allow load to be spread across all participants, which provides automatic load balancing compared to a centralized server handling all the load in the network.

P2P networks also allow new protocols to be developed for networking without the need to rewrite the networking stack on the operating system level. This is facilitated by the overlay nature of P2P systems. It can be argued that IP is fundamentally a P2P design; enabling peers in a network to route data to each other. P2P overlay networks may reveal new (and better) designs that can eventually be used at lower levels for routing. Of course, there are trade-offs which must be considered when designing any network; P2P networks work well in cases where decentralization, openness and scalability are required.

There are some drawbacks to P2P systems, such as the lack of trust between peers, the spread of malicious data and sometimes high latencies. Since peers often should not trust one another, additional security needs to be incorporated into P2P designs. Viruses and worms have been injected into P2P networks and disguised in order to get unsuspecting and overly trusting peers infected. Low resource peers participating in P2P networks can degrade performance for other peers. This makes it difficult or impossible to provide guarantees on data transfer rates or the wall-clock time required to perform a computation.

Definition 1.1 (P2P Network)**.** *A network of peers in which all peers participate in both providing and utilizing services.*

1.2.1 Centralized P2P

A centralized P2P system is one in which participants can freely join the network, but contact a central authority for certain required information. Probably the best example of a centralized P2P system used on a large scale is BitTorrent [32]. BitTorrent is a content distribution protocol that leverages the decentralized nature and low cost of P2P. With BitTorrent, a single distribution source (the central “authority”) releases content for download. As peers in the network download the content, they also upload parts of it to other peers. At the start, there is only a single source from which to download the content, but each downloading peer also acts as a source. In this way the original distributor need not provide the capacity in bandwidth for each peer that downloads the file, and the load is spread across all peers. BitTorrent traffic has been estimated to account for between 30 and 75 percent of all Internet traffic [162]; these numbers vary widely regionally, but BitTorrent certainly accounts for a significant portion of all Internet traffic today.

Another example of a centralized P2P system was Napster [62]. File sharing may have been the first mainstream use of P2P, and Napster [62] certainly brought it to the average computer user. Napster (in its original incarnation) was a system in which peers could join the network and share files with other peers. Napster was a P2P network because transfers were made between peers directly, but it was centralized because a central server was used to index which peers were storing which content.

Centralized P2P systems work well because they do not require a sophisticated routing algorithm. The indexing or controlling central server simply stores all the information required to connect peers, or parcels out jobs to be performed by the peers in the network. However, the centralized aspect of these networks can also be a problem as the central server responsibilities may not scale as new peers are added, and the central server may represent a single point of failure.

1.2.2 Pure P2P

A pure P2P system is a network in which peers can participate without the need for any central authority and peers are supposed to operate according to the same protocol. Pure P2P systems can be used for distributed computation, such as SETI@home [2] and Globus [66]. These allow networks of peers to pool processing power to solve computationally intensive problems. Pure P2P designs are used for diverse forms of communication including VoIP [186] teleconferencing, instant messaging [156] and on-line collaboration [52]. Other pure P2P systems include mobile sensor networks [96], ad-hoc wireless and mesh networks [174], TV [105, 127], and anonymity systems [190].

Pure P2P designs are the most decentralized P2P systems, as no peers in the network are required to have more resources or responsibility than any other peers. However, resource discovery can be more difficult in these networks, as well as connecting peers in an appropriate topology.

1.2.3 Super-peer P2P

A super-peer P2P system is one which combines elements of centralized and pure P2P designs. Super-peer designs are made up mostly of peers with relatively few connections. However, there are some peers that have many more connections than normal peers; these peers are known as super-peers or ultra-peers. The main advantage of these super-peers is that routing requires fewer hops due to the high level of connectivity.

A popular super-peer network is Gnutella [49], a file sharing design that has seen widespread use [7]. Average Gnutella peers have approximately 3 connections; super-peers in the network by design have around 32 peer connections [49]. The main disadvantage of a super-peer design is that the super-peers are required to have more connections and provide more resources than other peers. The popular VoIP software, Skype, utilizes super-peers to provide connectivity for users behind NAT. In December of 2010, a software update was released which caused communication with these super-peers to fail. As a result, the number of Skype users plummeted from 23M to less than 2M over the course of hours [135]. While the outage lasted less than a few days for most users, this underscores how important a few key peers are in super-peer networks.

Super-peers also raise questions about the security of users of the P2P network. Malicious participants in the network could "volunteer" to become super-peers and conceivably monitor the actions of a large number of users in the network. This ranges from benign behavior such as monitoring the size of the network to malicious goals like a government determining dissidents in order to persecute them. While these actions are possible in any P2P network (depending on the security of the design), super-peers provide obvious targets for attacks.

1.3 Design Goals

Our design is a pure P2P system due to our focus on openness and decentralization. In this section we outline common goals for P2P systems, and explain which of these goals we focus on with our design and which goals are less important.

Definition 1.2 (Efficient Routing)**.** *A P2P routing algorithm is* ***efficient*** *if routing can be performed sub-linearly; as the number of peers in the network (n) increases, the number of operations per request increases sub-linearly (i.e. $O(\sqrt{n})$, $O(\log n)$, etc.).*

A specific concern of DHT designs, and P2P routing algorithms in general is efficiency. Efficiency is defined in terms of the number of nodes a request traverses before routing terminates. This measure is referred to as the number of "hops" a request must travel before termination. We relate the number of hops that a query takes to the number of nodes in the network n; therefore efficiency also encompasses scalability.

The best performance achieved in other DHT networks [112,143,173,187] is $O(\log n)$, the theoretical best case performance [139] with a routing table of size $O(\log n)$. More efficient networks (such as [49, 62]) typically rely on a centralized authority, "super peers" or routing tables with greater than $O(\log n)$ entries. Networks with a hierarchical structure are also able to achieve better efficiency than DHT networks [111], but these networks require peers at the "top" of the hierarchy to handle more requests than those "lower" levels. DHT routing algorithms which achieve routing in $O(\log n)$ steps require universal connectivity, so that routing tables can be constructed optimally. This routing table construction creates a highly structured network which enables efficient routing. However, these algorithms perform much worse in restricted-route topologies, where the desired network structure is impossible to achieve.

Definition 1.3 (Decentralization)**.** *A **decentralized** [93] routing algorithm operates only using local knowledge; including immediate neighbors, the route traversed so far, and the relation of the target of the request to the overall structure.*

Decentralization is an important goal for our system, so that it will work in a self forming manner and not be reliant on any trusted third party, or give certain peers more control over the network than others. We will consider our system to have fulfilled the goal of decentralization if all peers are equally distrusted and responsibility for requests and data storage is evenly spread amongst those peers.

Our network design must not assume any form of centralized control, meaning there is no way to authenticate peers in the network, nor is there a central point of contact that *all* peers can communicate with. This restriction comes from two points. First, centralization either requires a trusted third party or some system of distributed trust (which requires too much overhead to maintain and generally needs a routing algorithm to begin with). Second, in a restricted-route network peers may be participating in the network but unable to directly communicate with a central authority. More generally, two arbitrary peers can not be assumed to be able to communicate directly in a restricted-route network.

A subset of goals important for our design are security related. For instance, in our design, we leave the problems of authenticity (data verification), integrity (data cannot be altered) and confidentiality (data cannot be viewed in transit) to higher-level protocols [17, 30]. Common security goals for routing algorithms encompass the rest of our definitions.

Definition 1.4 (Reliability)**.** *A routing algorithm is **reliable** if a route to the intended target is found with high probability provided such a route exists (and does not include malicious peers).*

Reliability is arguably the most important property of a routing algorithm; as such, it is crucial that our algorithm be reliable. Many DHT designs are unable to provide guarantees that data stored in the system will be retrievable by other peers with high probability. This limitation is caused by two main problems, either the route through the network fails to find the proper peer, or the peer responsible for the data is malicious.

Definition 1.5 (Fault-tolerance)**.** *A system is byzantine [97]* ***fault-tolerant*** *if it functions in the presence of arbitrary behavior by participating peers.*

Due to the openness and lack of trust inherent in our design, we must make our system as fault tolerant as possible. This means that our system needs to be resistant to DoS attacks and misbehaving peers. There are many different kinds of attacks which must be considered, including those described in Chapter 2 and Chapter 4.

We need to consider how malicious participants or adversaries affect the network, and limit the harm they are able to do. This is a fundamental aspect of fault-tolerance, as the open nature of P2P networks makes them easy targets for attacks. A fault-tolerant algorithm should not allow malicious peers to use more resources in the network than non-malicious participants. Also, requests should not able to be crafted such that they allow asymmetric resource usage. Asymmetric resource attacks are possible in some P2P networks, and are easy methods for performing denial of service (DoS) attacks.

Definition 1.6 (Availability)**.** *A DHT provides good* ***availability*** *if data previously stored in the network is returned with high probability.*

Availability is a fundamental requirement for our design. Perfect availability is impossible, but provided that our algorithm finds data with high probability (for some measure of "high probability") is sufficient for our goals. Availability also concerns distributing data across the network so it cannot be lost unless a significant portion of peers go off-line, meaning data remains in the network when nodes go off-line.

Definition 1.7 (Consistency)**.** *A routing algorithm provides* ***consistent*** *performance if performance data shows little variation.*

Little variation in this case generally translates to a small standard deviation in relation to the average value. A routing algorithm that provides consistent performance is desirable because bandwidth and latency estimates can be easily obtained. This is generally achievable if the routing algorithm is deterministic, and identical requests will be routed the same way each time. Consistency is not the focus of our routing algorithm, because we have chosen other goals such as decentralization and fault tolerance to be of greater importance.

Definition 1.8 (Anonymity)**.** *A routing algorithm provides* ***anonymity*** *if the mapping between operations and the peer that performed those operations cannot be determined.*

Anonymity is not currently a property that we have integrated into our routing design. Though not precluded from our design, it is not essential for our system. Providing strong anonymity is difficult, as shown in Chapter 4. We incorporate some of the techniques of onion routing [71, 177] as a part of our design, described in Chapter 5. Our usage of onion routing is not intended to provide anonymity, though it could potentially be used to do so in the future.

Definition 1.9 (Data Integrity)**.** *Data that is found in the DHT can be verified as the correct data that was searched for.*

There is inherent difficulty in guaranteeing that data returned by a routing algorithm is the *correct* data. For instance, the same "malicious" file to one user may be the botnet kit searched for by another user. While this is an important goal, it is problematic to provide at a low level, so we do not focus on this goal at the routing algorithm level. High-level content verification can be better provided by applications which use the low-level DHT routing algorithm.

1.4 Methodology

So far we have described the design goals for a secure, decentralized DHT routing algorithm for P2P networks. We describe our implementation in detail in Chapter 7, but instead of jumping directly into the design and implementation it is important to describe our methods for evaluation of other protocols and systems in this domain. The general approach we have is as follows: identify P2P systems that are in use that have requirements similar to our own, study these designs from a security perspective, identify any problems or flaws with these designs or implementations and evaluate those issues for practical usage. By doing so, we are able to identify important factors that we then take into consideration with our design and implementation. Evaluations which reveal some of these factors are the topic of Chapter 2 and Chapter 4. We perform evaluations by performing simulations or attacks on the real network based on the flaws we discover. In some cases, specifically Chapter 3 and Chapter 5, we identify an extant problem with the underlay topologies our P2P design is meant for. These chapters provide background and steps towards mitigating these problems, which enable our routing algorithm to perform properly. We then move on to an explanation the framework for our design evaluation in Chapter 6, then finally present the design and analysis in the Chapter 7.

The next chapter details the difficulty in making a secure routing algorithm for restricted-route networks. This shows the importance of considering design goals, previous work, and potential adversaries and attack vectors when designing a routing algorithm.

1.5 Summary and Overview

P2P systems and the applications built upon them make up a large portion of the usage of the Internet today [162], yet they tend to lack good security and efficiency or trade off one for the other. There is a demand for a secure and efficient routing design which works well in the kind of networks used today, which we provide in this thesis. The Internet, while a large and mostly robust system, has a number of problems such as DNS poisoning and route hijacking through BGP vulnerabilities that may never be completely fixed. There is a need for a routing algorithm that works well both on the current NAT restricted Internet as well as for other uses, such as in ad-hoc networks. Before presenting our P2P routing algorithm which we provide as a partial solution to these problems, we describe our research evaluating other P2P routing algorithms. We also detail some of the work we have done to reduce the effect of NAT and restricted route networks on P2P overlays in general. The next chapter describes our detailed analysis of a first attempt at the creation of a self organizing P2P overlay network for use in restricted-route networks, and why this solution was found to be lacking in some of our design goals.

2. ROUTING IN THE DARK: PITCH BLACK

This chapter introduces Sandberg's Freenet 0.7 routing algorithm, which provides efficient routing in restricted-route networks. Before creating our own routing algorithm design, we sought to evaluate extant designs that already claimed to provide solutions to the problem. After a detailed analysis, we found that the Freenet routing algorithm does not provide an acceptable solution to the problem, because a small number low-cost attackers are able to severely disrupt the operation of the network. However, analyzing Sandberg's routing algorithm has provided key insights into pitfalls to avoid when designing our own. This Chapter is based on work previously presented at ACSAC 2007 [59], and also the author's master thesis [57].

2.1 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 even 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 [33,72,112,143,173]. However, all of these designs are unable to work 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. Such restrictions arise from a variety of sources, such as physical limitations of the communications infrastructure (wireless signals, physical network topology), policies (firewalls) or limitations of underlying protocols (NAT, IPv6-IPv4 interaction).

Recently, a new routing algorithm for restricted-route topologies was proposed [158] and implemented in version 0.7 of Freenet, an anonymous peer-to-peer file-sharing network [30]. 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.

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

The research presented in this chapter shows that any participating node in the Freenet network can severely degrade the performance of the routing algorithm by changing the way it behaves during the location swapping portion 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 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 why churn impacts the structure of the overlay negatively, a phenomenon that was observed by Freenet users in practice but has, to the best of our knowledge, never been explained.

The chapter is structured as follows. Section 2.2 describes related work focusing on distributed hash tables and Small-World networks. Section 2.3 details Freenet'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 Section 2.4, followed by experimental results showing the effects of the attack in Section 2.5. Possible defenses and their limitations are discussed in Section 2.7.

2.2 Related Work

2.2.1 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 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.

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 [141]. 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

(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

Tab. 2.1: Performance metrics for DHTs.

	Chord [173]	Pastry [153]	Kademlia [112]	CAN [143]	RSG [72]
(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

Tab. 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.

that it is generally possible for any peer to connect to any other peer. However, firewalls and NAT make this assumption unrealistic over the current Internet, where large-scale studies have shown that over 70% of machines are NATed [27].

In contrast to the DHT designs from Table 2.2, the Freenet routing algorithm achieves expected $O(\log n)$ routing in restricted-route topologies under the assumption that the restricted network topology has Small-World properties.

2.2.2 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 [183], the most prominent example being social networks [115].

Watts and Strogatz [183] 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' work explains why short paths exist in real-world networks.

Kleinberg [92,93] 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's "darknet" routing algorithm.

2.3 Freenet's "darknet" routing algorithm

Freenet [30] is a peer-to-peer network where the operator of each node specifies which other peers are allowed to connect to the node. 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 commonly arise given a certain amount of "randomness" in the graph construction, it is realistic to assume that Freenet's darknet is a Small-World network. The routing restrictions imposed on the Freenet 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.

2.3.1 Network creation

The graph of the Freenet 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 assumes that a sufficient number of edges (or friend relationships) between peers will exist so that the network will be connected.

Each Freenet 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 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.

2.3.2 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 [159]. Figure 2.1 shows the routing algorithm for *GET* operations in pseudo-code. A *PUT* operation is routed in the same fashion and reaches exactly the same peers as an unsuccessful *GET* operation. In addition, Freenet 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 node's 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.

2.3.3 Location swapping

To make the routing algorithm find the data faster, Freenet 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.

$$D_1(A,B) = \prod_{(A,n)\in E} |L(A) - L(n)| \cdot \prod_{(B,n)\in E} |L(B) - L(n)| \qquad (2.1)$$

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:

$$D_2(A,B) = \prod_{(A,n)\in E} |L(B) - L(n)| \cdot \prod_{(B,n)\in E} |L(A) - L(n)| \tag{2.2}$$

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 [158] shows that the Freenet 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 [72, 112, 153, 173].

Fig. 2.1: Pseudo-code 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.

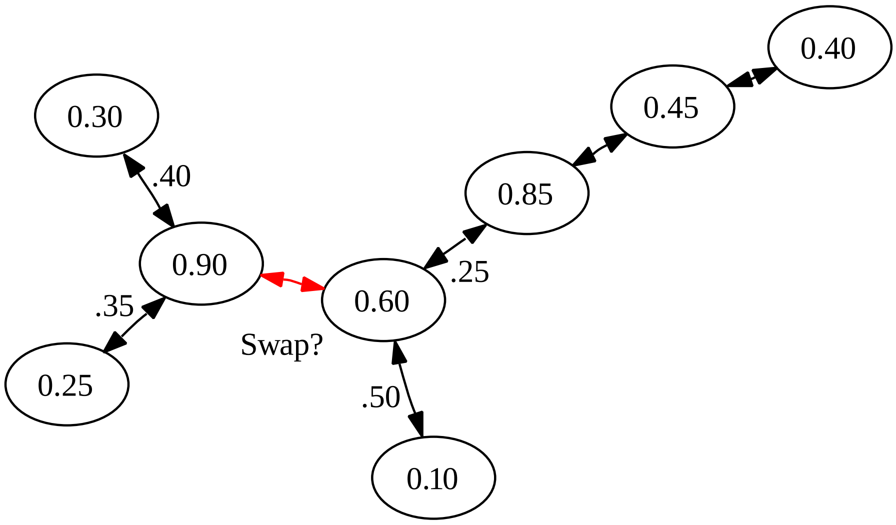
5. Find the closest neighbor (in terms of peer location) with respect to the key of the *GET* request, excluding those routed to already. Forward the request to the closest peer with a (probabilistically) decremented hops-to-live counter. If valid content is found, forward the content to sender, otherwise,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

2.3.4 Content Storage

Each Freenet node stores content in a datastore of bounded size. Freenet 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.

2.3.5 Example

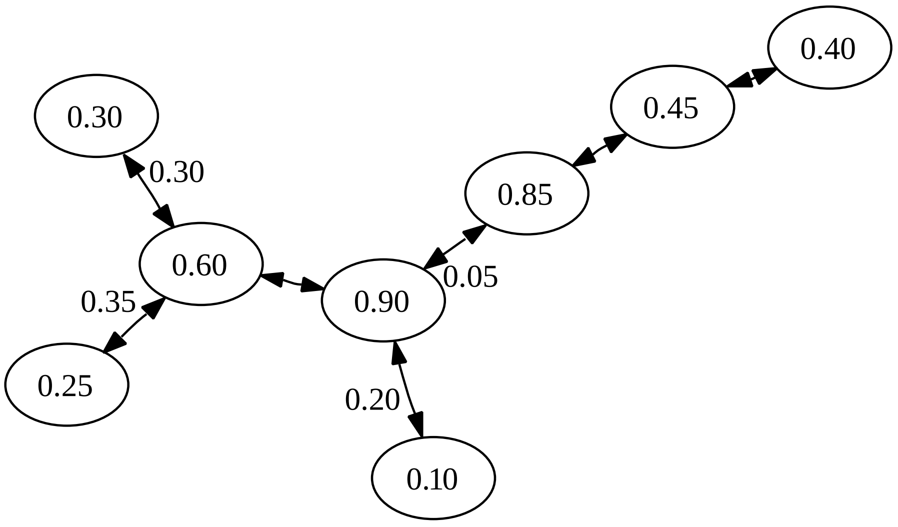
Figure 2.2a shows a small example network. Each node is labeled with its location ($L_n \in [0, 1)$) in the network. Bi-directional edges indicate direct connections between nodes. In a friend-to-friend network, these are the connections that were specifically allowed by the individual node operators, 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, as the algorithm works for any connected graph.



(a) An example network with two nodes considering a swap. The result of the swap equation is $D_1 = .60 * .65 * .25 * .50 = .04875$ and $D_2 = .30 * .35 * .05 * .80 = .0042$. Since $D_1 > D_2$, they swap.

Fig. 2.2

The network illustrated in Figure 2.2a happens to have an assignment of locations 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 (2.1). Figure 2.2b shows the new assignment of locations after the swap. Note



(b) Post-swap

Fig. 2.2: This figure shows an example network before and after a swap occurs.

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 traversal during routing.

Figure 2.3 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 a “content not found” message. 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 2.4 illustrates how Freenet 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 the closest node (0.90 in this case) is found, the request is sent to all neighboring nodes. These neighbors do not forward the request (since 0.90 is closer to the key than they are), thus ending the routing process.

2.4 Security Analysis

The routing algorithm works under the assumption that the distribution of the keys and peer locations is random. In that case, the load is 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.

The basic idea behind the attack is to de-randomize the distribution of the node locations. 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 clusters are responsible for less content (because many other nodes are also close to the same set of keys), whereas the load for nodes outside of the clusters is disproportionately high.

We will now detail two scenarios which destroy the initial random distribution of node locations resulting in the clustering of locations around particular values. The first scenario uses attack nodes inside the network. This attack quickly unbalances the load in the network, causing 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

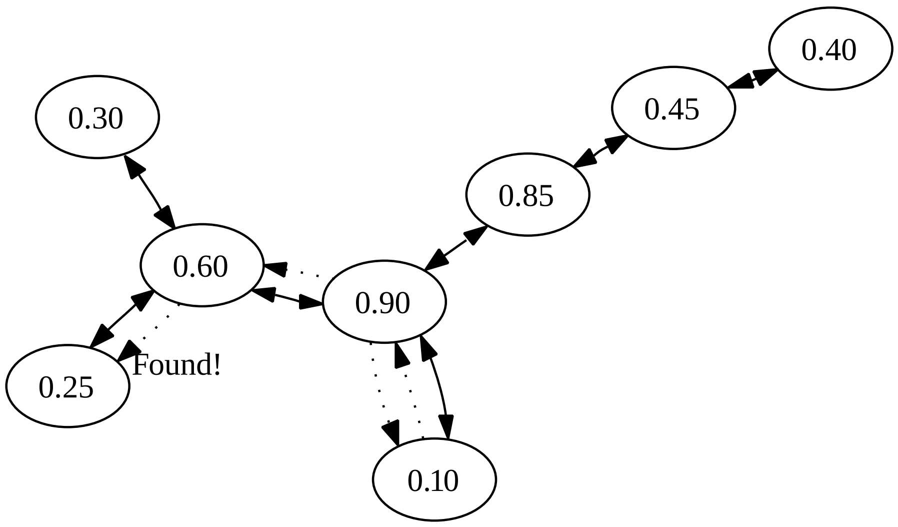
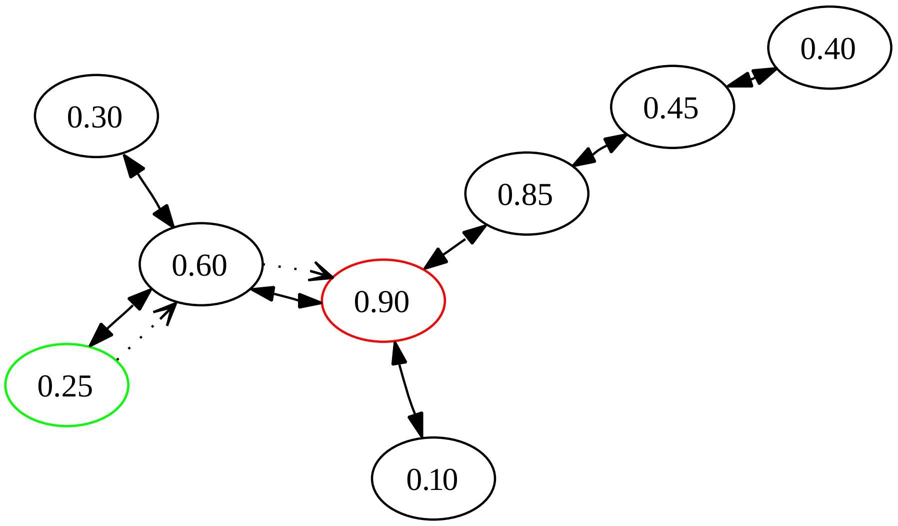


Fig. 2.3: Illustrates the path of a *GET* request initiated from the node with location of 0.90. The request is looking for data with a key value of .23, which is stored at the node identified by the location 0.25. The path that the *GET* request travels is displayed as the dotted lines which travel from 0.90 $\rightarrow$ 0.10 $\rightarrow$ 0.90 $\rightarrow$ 0.60 $\rightarrow$ 0.25 where the data is found.



(a) 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 → 0.60 → 0.90, where the data is stored.

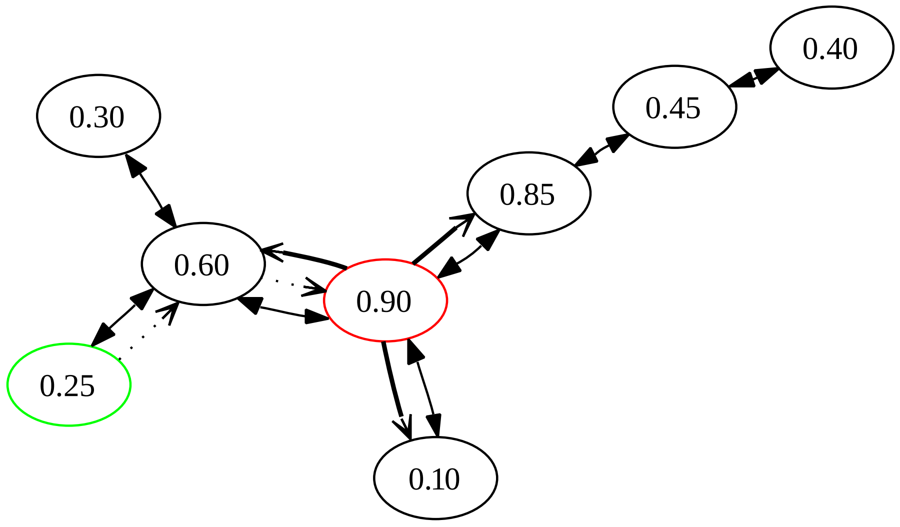
Fig. 2.4

nothing. The second scenario illustrates how location imbalance can occur naturally even without an adversary due to churn.

2.4.1 Active Attack

As described in Section 2.3.3, a Freenet node attempts to swap 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 (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 from swapping. 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 $L(N) = m$ afterwards. 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 (2.1). 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 protocol. Again, the F2F topology prevents the neighbor involved in the swap from checking the validity of this information. After the swap, the attack node can again assume the original location m and



(b) Result once a *PUT* has reached 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.

Fig. 2.4: The graph on top illustrates the

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 location. Once the location has been spread, the adversary subjects another neighbor to a swap, removing yet another random location from the network. Figure 2.5 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. Thus, a trade-off exists between the speed of penetration and the impact of the attack in terms of causing load imbalances.

2.4.2 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 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

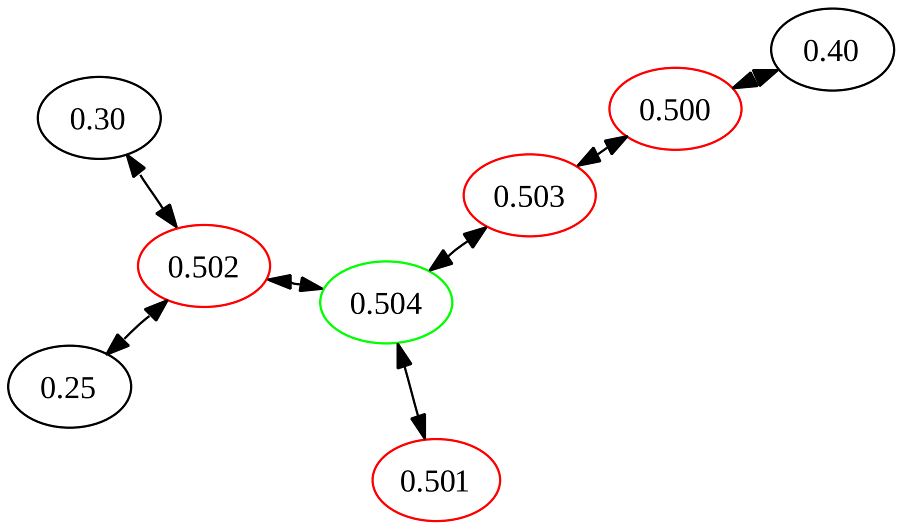


Fig. 2.5: This figure shows the example network after a malicious node has started to spread locations close to 0.5 by swapping. In this figure the malicious node currently has location = 0.504

random locations. This section illustrates that the opposite is the case: natural churn can strengthen the attack and even degenerate the Freenet 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'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 disrupt network operations. Since honest Freenet 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'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 overlay. This additional impact requires that 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 (2.1)). 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, 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 2.4.1 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 users and was reported to the Freenet developers – who so far have 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 (Section 2.5) hold for both scenarios.

2.5 Experimental Results

This section 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 [92]) 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

[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.

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 “super-nodes”, and a Small-World graph augmented with super-nodes. Super-nodes are common in some networks such as [49] 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.

2.5.1 Distribution of Node Locations

Figures 2.6a, 2.6b, 2.6c and 2.6d illustrate the distribution of node locations on a circle before, during and after an attack. The initial distribution in Figure 2.6a 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 2.6b, 2.6c and 2.6d 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 2.6b shows the node locations at an early point in the attack, after the 15^{th} attack iteration (90 total iterations). Even so, the clustering effect of our location swapping attack can be seen. In 2.6c 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 2.6d 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.

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, but 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 plotted on the circle has an x, y coordinate. The Cartesian distance from each 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 threshold, that point's "size value" is incremented. To keep the points from growing out of control the $\log^2$ of each size value is used as a multiplier applied to the base size of the points 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

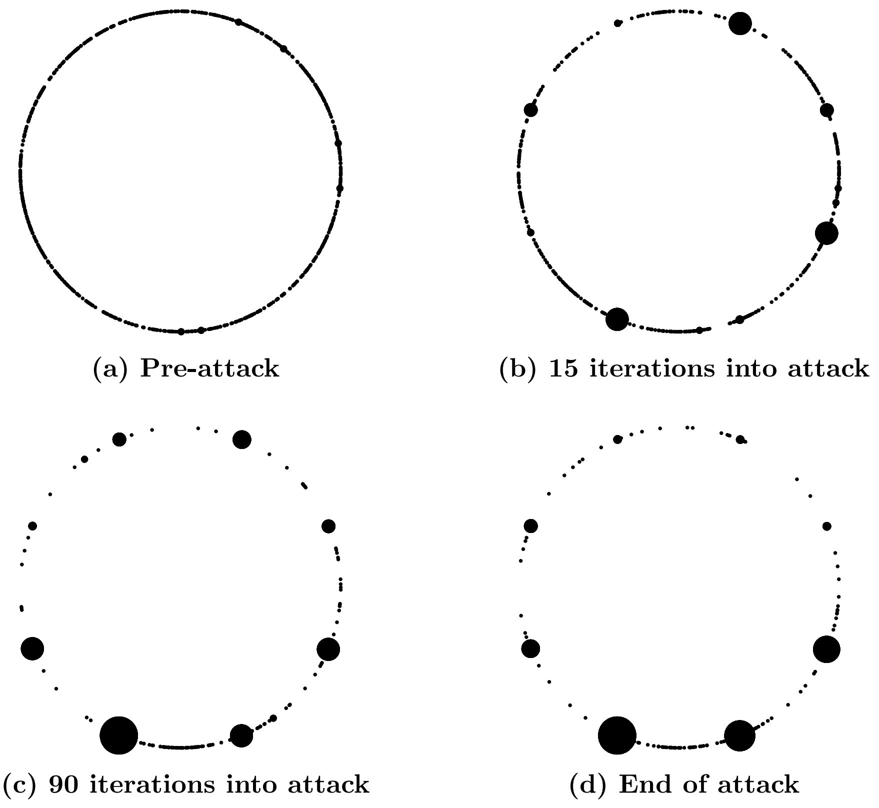

(a) Pre-attack
(b) 15 iterations into attack
(c) 90 iterations into attack
(d) End of attack

Fig. 2.6: Plot of node locations on a circle representing the Freenet address space for an 800 node Freenet network. Dot size is scaled to show the number of nodes clustered at particular locations. Over time, the attack successfully clusters nodes at the 8 attacker locations.

manner on the 400 node test network, this time using 16 malicious nodes with 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 2.7a shows the initial plot of node locations, evenly spread over the range of possible values. Figure 2.7b shows the picture after 25 iterations of the attacked network, and again the effect of the attack is readily apparent, and Figure 2.7c, showing the result of 50 iterations of attack time tells the same story. Figure 2.7d 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 2.8, which plots the number of poisoned nodes against the trial iteration. It is clear that shortly after the attack begins in the 25^{th} iteration, swaps are forced with all of the malicious nodes immediate neighbors. After this spike, malicious node locations can only be spread by the normal swapping

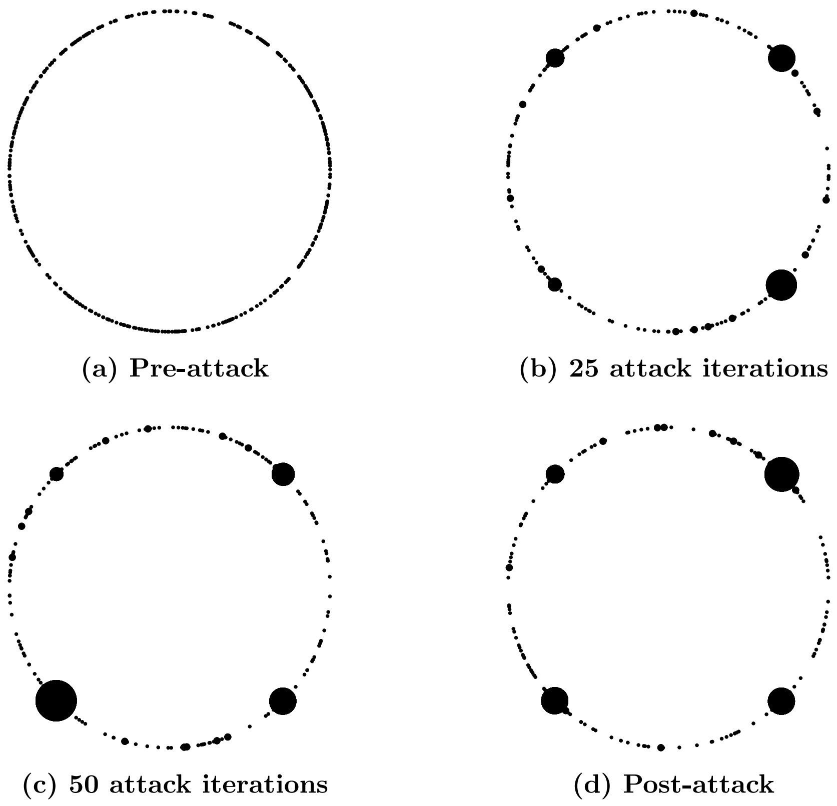


Fig. 2.7: Plot of node locations on a circle representing the Freenet address space for a network of 400 nodes. Dot size is scaled to show the number of nodes clustered at particular locations. Over time, the attack successfully clusters nodes at the 4 attacker locations.

algorithm; therefore the number of malicious locations increase more slowly, but steadily throughout the attack duration.

Table 2.3 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.

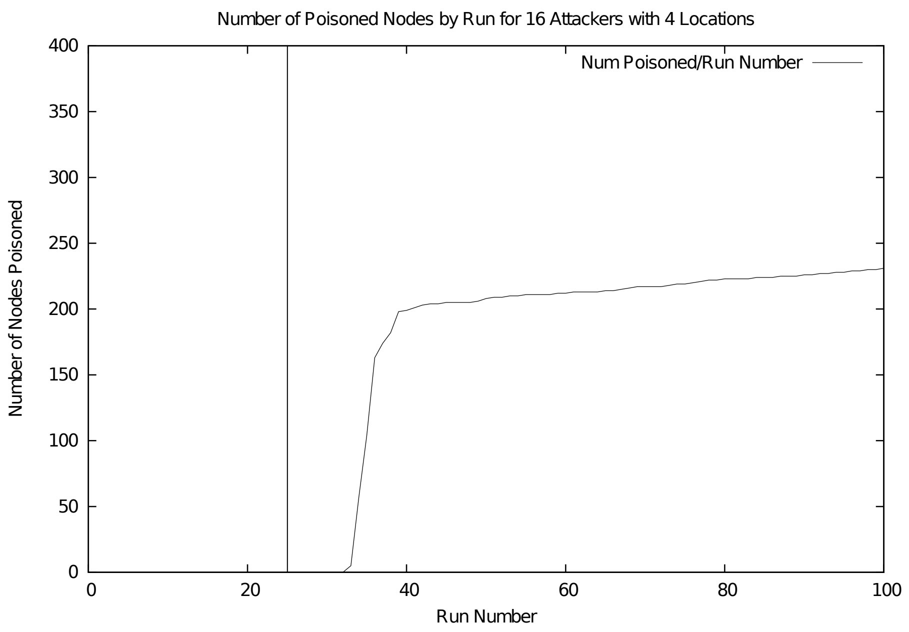


Fig. 2.8: Plot of the number of nodes that have been "poisoned" by run number.

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.

2.5.2 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 metrics for determining the number of hops. 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 higher 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 2.9 show an example of this metric for a 400 node network where the attack begins at the 25th 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,

# 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

Tab. 2.3: Data showing average number of poisoned nodes for varying configurations of our 400 **node testbed and different points in time.**

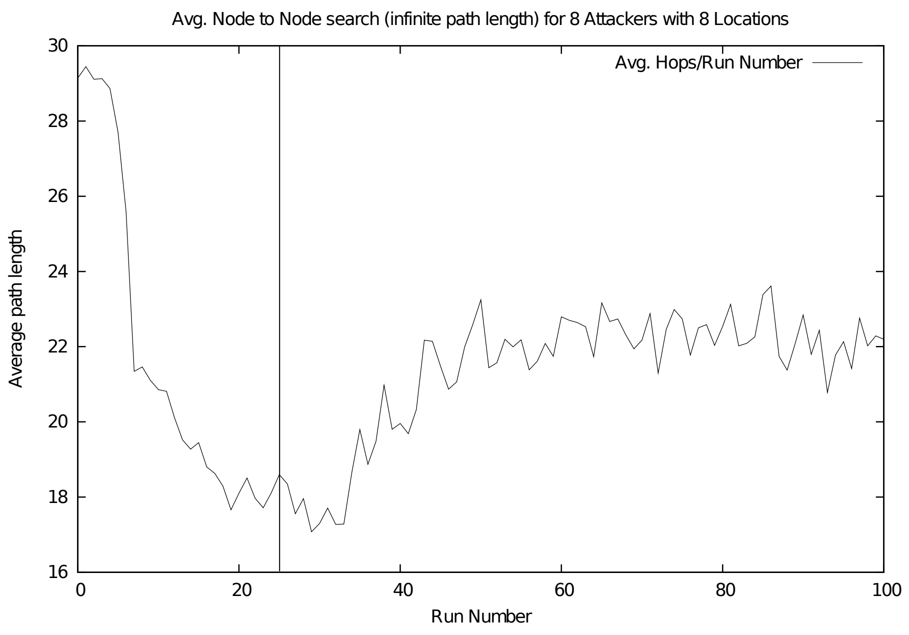


Fig. 2.9: Graph showing average path length of completed requests when routing is unbounded.

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 2.3, 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 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. Figures 2.10 and 2.11 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

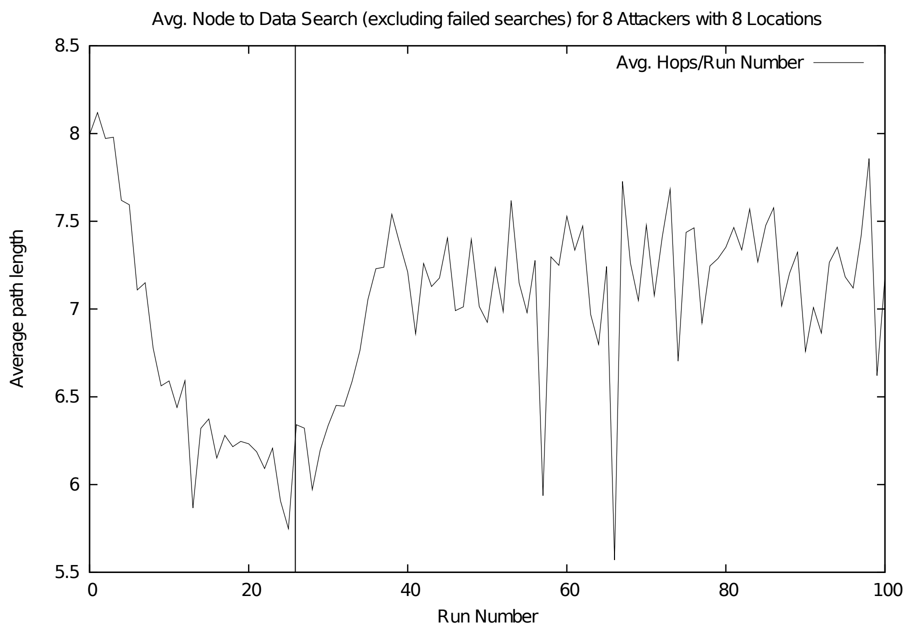


Fig. 2.10: Graph showing average path length from source to destination of completed requests.

low number of hops and convergence decreases the hops, our attack nearly removes all the gains of this convergence.

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.

2.5.3 Availability of Content

Figures 2.12a, 2.11b and 2.10c 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.

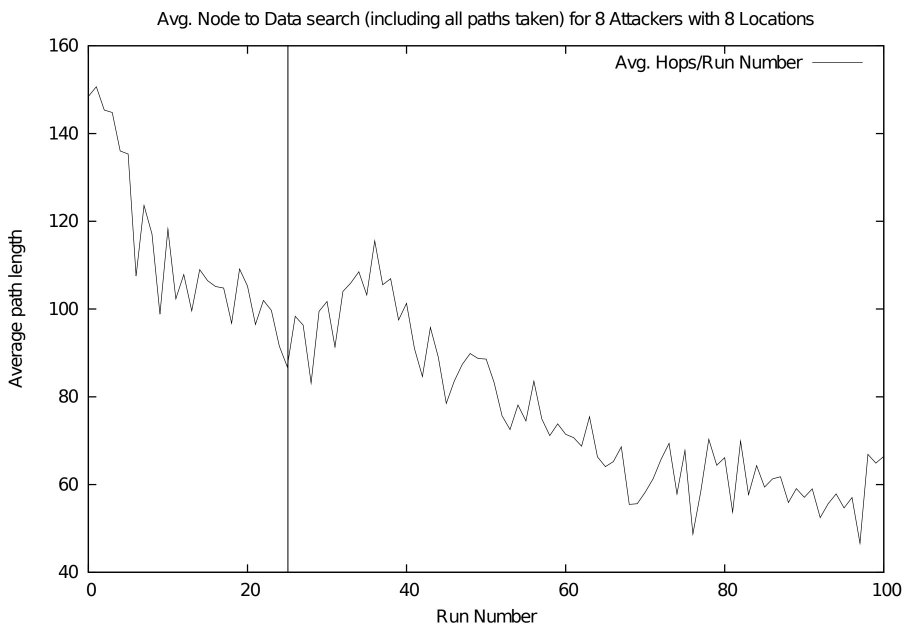
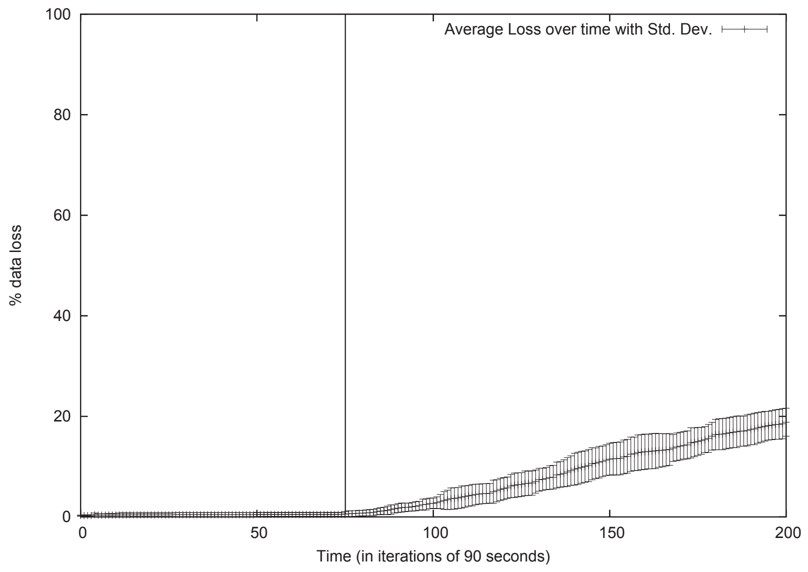


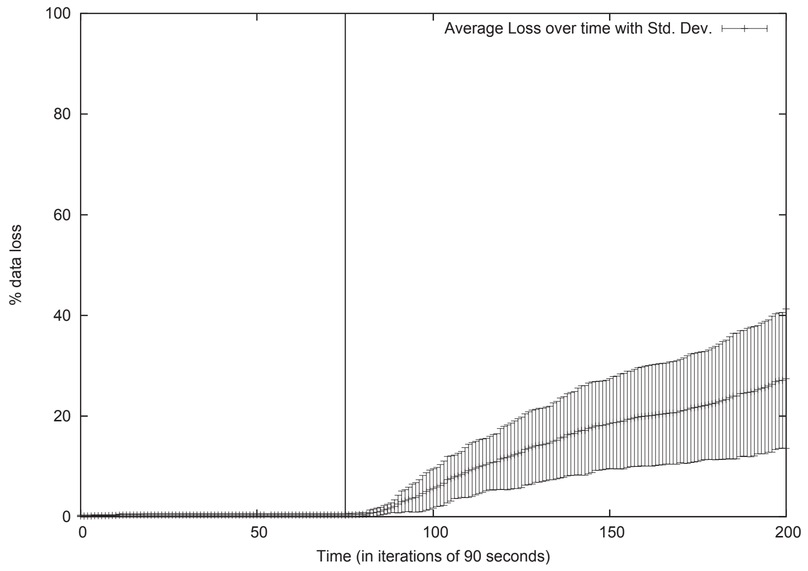
Fig. 2.11: Graph showing average cumulative path length including all nodes traversed of completed requests.

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.

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 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

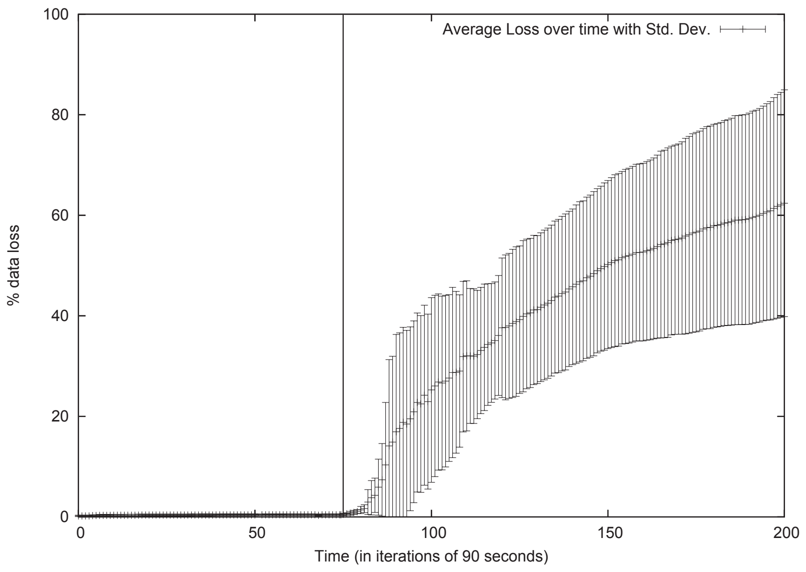


(a) Graph showing average data loss over 5 runs with 800 total nodes and 2 attack nodes using 8 attacker chosen locations with the attack starting at iteration 75 **(horizontal line depicts attack start time).**



(b) 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).**

of data. As discussed in Section 2.5.2 the length the paths needed in order to find data in the network also plays an important part; if the path to the



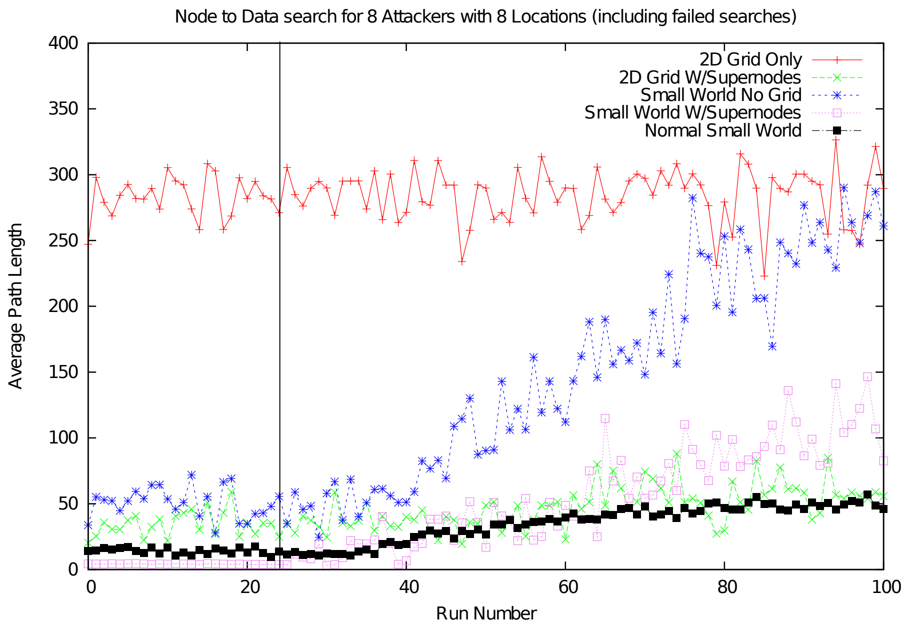
(c) Graph showing average data loss over 5 runs with 800 total nodes and 8 attack nodes using 8 attacker chosen locations with the attack starting at iteration 75 (horizontal line depicts attack start time).

Fig. 2.10: These figures show the increased success of the attack when using more attacker nodes. Note that the final graph still has only 1/100th of the total nodes as attackers.

data is too long the query will still fail even if the data is present in the network.

2.5.4 Other Topologies

Though the Freenet 0.7 routing algorithm theoretically works best in 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 2.5.2 for each of the topologies presented. 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 Cartesian 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 2.3.1 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.



(a) Average routing hops for queries when failed searches are assumed to reach all nodes.

Fig. 2.11: Results from multiple topologies for different hop-count metrics.

Our other two topologies use the concept of “super-nodes” [49], where certain nodes exist in the network that are much more highly connected (or universally connected) than “normal” nodes. We implemented super-nodes very simply in our topology construction; when enabled, we selected 1 percent of the total nodes probabilistically to be super-nodes, 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 super-nodes in two scenarios, with our normal Small-World topology and with the simple 2d-grid topology.

The graphs in this section each have 5 lines, one for each topology and are labeled as such. For consistent comparison between each topology, all results are from graphs with 400 nodes, 8 malicious nodes which are switched into attack mode at iteration 25 with the malicious nodes rotating through 8 locations over the course of the attack. The figures follow the same progression as those from Section 2.5.2.

Figure 2.11 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 super-nodes are shown to be the least effected by our attack. The 2d-grid with super-nodes is relatively the same across the board, which means that convergence due to the swapping algorithm and our attack have little effect on routing. The Small-World with super-nodes 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.

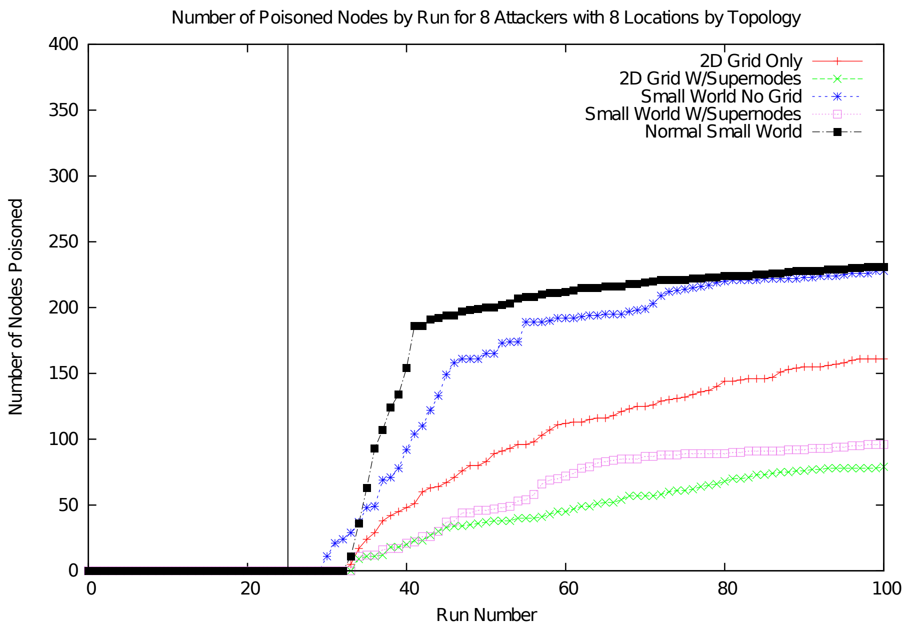
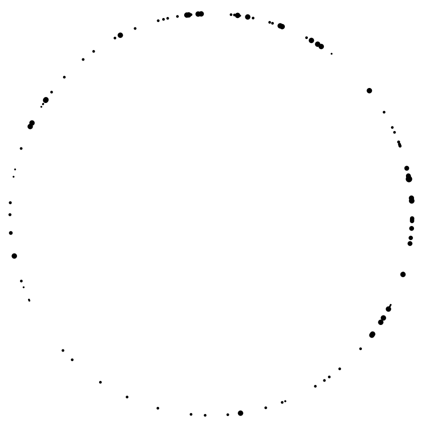
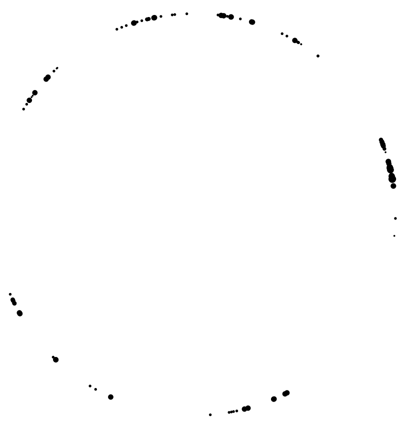


Fig. 2.12: Number of poisoned nodes over time for multiple topologies.

The final figure in this section shows the results of the number of poisoned nodes in the network as the attack progresses. This data, shown in Figure 2.12, is basically what we 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 also shows a high number of poisoned 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 super-nodes shows the least number of poisoned locations, though still close to $\frac{1}{4}$ of all locations have been subverted. The most surprising result is that of the low


(a) 0 rounds of churn

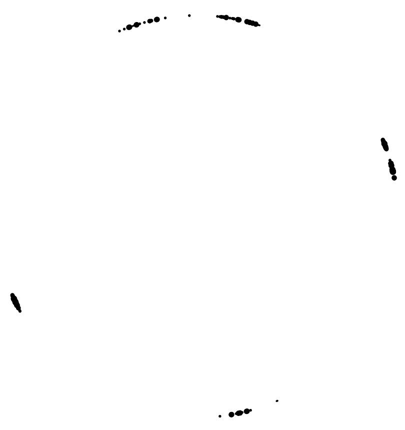

(b) 100 rounds of churn

number poisoned for the Small-World with super-node topology. However, the growth remains steady which is the only real requirement for us to prove that our attack works in various topologies.

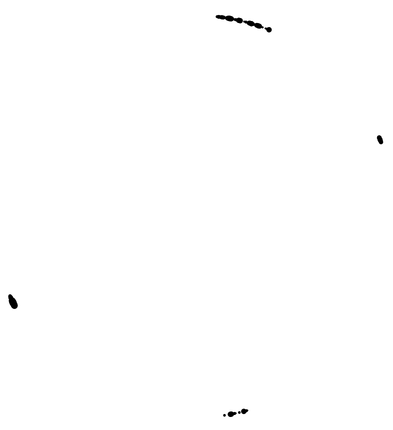
2.6 Simulation of Churn

Figure 2.11 shows 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 (as described in Section 2.4.2). In each round of simulated churn, ten of the remaining one 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.

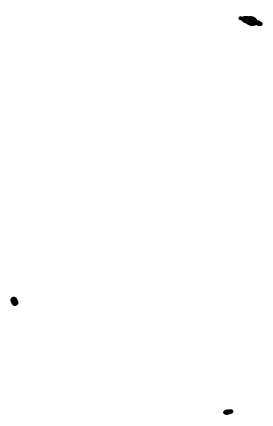
These results are particularly interesting because they show that the kind of location clustering which is produced by our attack will also happen in a network with no attackers, but has a stable core of peers and experiences churn. This is actually a very likely situation for any peer-to-peer network,



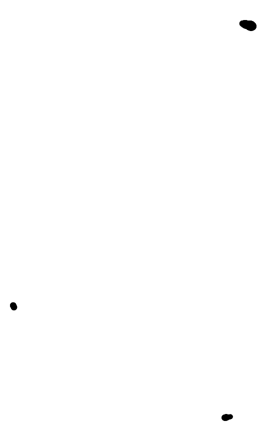
(c) 2,000 rounds of churn



(d) 10,000 rounds of churn



(e) 50,000 rounds of churn



(f) 100,000 rounds of churn

Fig. 2.11: Plots of node locations on a circle representing the Freenet address space. Each figure shows the stable core of 500 nodes in the same network after varying amounts of churn; the result of which is clustering similar to our attack result.

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 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 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 [175]. 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.

2.7 Discussion

Various techniques have been proposed that may be able to limit the impact of the attack described in this chapter, including changing the swapping policy, malicious node detection, and secure multi-party 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 here 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 carefully. In order to accurately choose 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 Section 2.5, 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 node 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 (with respect to the data) in the network. Resetting a node's location 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 the network indefinitely it needs to be refreshed periodically. However, if all nodes reset their locations in a short time period the network could be flooded with data being as the nearest peer in the network is constantly changing.

Another possible method for mitigating our outlined attack involves each peer determining malicious nodes 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 likely that nodes have wide variation in their number 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 neighbors 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 reasons 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 multi-party 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 direct neighbors (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 disseminating 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 of that nodes participation in the network.

2.8 Conclusion

The new Freenet routing algorithm is unable to provide reasonable availability guarantees in networks where adversaries are able to participate. The algorithm also degenerates over time (even without active adversaries) if the network experiences churn. The recommended approach to address both problems is to periodically reset the locations of peers. While this limits the deterioration of the routes through adversaries and churn, such resets also sacrifice the potential convergence towards highly efficient routes. Secure and efficient routing in restricted-route networks remains an open problem.

In the scope of this thesis, this chapter provides unique insight into a key problem when designing a routing algorithm for restricted-route topologies. Specifically, we need to be very careful in what information nodes use to make decisions about routing. If any of that information comes from untrusted (or untrustworthy) sources it needs to be handled very carefully. As we describe in Chapter 7, our routing algorithm uses no information obtained from other peers directly. However, other peers may be used for routing if the fish-eye bounded distance vector transport is used. This transport facilitates efficient routing in topologies where the base topology is too sparse for our routing algorithm. Distance vector can allow virtual connections between distant peers even in physical networks. The issues inherent in the distance vector design, and protections against attacks based on it are provided in Chapter 5. Commonly, network topologies on the Internet are restricted by NAT, which also can make routing tables too bare for our routing algorithm to perform well. For this reason,in the next chapter we explore a method we designed and implemented for providing additional connectivity options for peers behind NAT devices.

3. AUTONOMOUS NAT TRAVERSAL

From a P2P networking perspective, and more specifically for our DHT routing algorithm operating in restricted-route networks; the goal of NAT traversal techniques is to provide a greater number of direct connections to peers than would otherwise be possible. Many techniques exist for NAT traversal; and the framework in which we have implemented our DHT routing algorithm employs as many of these techniques as possible to provide better connectivity to users. The framework implementation is out of the scope of this thesis; however the novel method presented in this chapter follows closely the design goals of our routing algorithm. It operates without the help of a trusted third party, and increases the chances of success of providing connectivity to peers behind NATs along with other NAT traversal techniques. The work presented in this chapter was previously published in the proceedings of P2P 2010 [122].

3.1 Introduction

Traditional NAT traversal methods require the help of a third party for signaling. This chapter investigates a new autonomous method for establishing connections to peers behind NAT. The proposed method for autonomous NAT traversal uses fake ICMP messages to initially contact a NATed peer. In this chapter we also present the theoretical basis for the method, discuss some possible variations which may provide better results in certain scenarios, introduce various concrete implementations of the approach and evaluate empirical results of a measurement study designed to evaluate the efficacy of the method in the real world.

A large fraction of the hosts in a typical peer-to-peer network are in home networks. Most home networks use network address translation (NAT) [53] to facilitate multiple computers sharing a single global public IP address, to enhance security or simply because the provider's hardware often defaults to this configuration. Recent studies have reported that up to 70% of users access P2P networks from behind a NAT system [27]. This creates a well-known problem for peer-to-peer networks since it is not trivial to initiate a connection to a peer behind NAT. In this chapter, we will use the term *server* to denote a peer behind NAT and the term *client* for any other peer trying to initiate a connection to the server.

Unless configured otherwise (protocols such as the Internet Gateway

Device Protocol [86] are counted as configuration in this context), almost all NAT implementations refuse to forward inbound traffic that does not correspond to a matching recently issued outbound request. This is not primarily an implementation issue: if there are multiple hosts in the private network, the NAT is likely unable to tell which host is the intended recipient. Configuration of the NAT is not always an alternative; problems range from end-user convenience and capabilities of the specific NAT implementation to administrative policies that may prohibit changes to the NAT configuration (for example, due to security concerns).

Since NAT systems prohibit inbound requests that do not match a previous outbound request, all existing NAT traversal techniques (aside from those require a configuration change to the NAT box) that we are aware of require some amount of active facilitation by a third party [151, 152]. The basic approach in most of these cases is that the server in the private network behind the NAT is notified by the third party that the client would like to establish a connection. The server then initiates the connection to the client. This requires that the server maintains a connection to a third party, that the client is able to locate the responsible third party and that the third party acts according to a specific protocol.

The goal of the method described in this chapter is *autonomous* NAT traversal, meaning NAT traversal without reliance on a third party. Using third parties increases the complexity of the software and potentially introduces new vulnerabilities. For example, if anonymizing peer-to-peer networks (such as GNUnet [16] or Tor [45]) used third parties for NAT traversal, an attacker may be able to monitor connections or even traffic volumes of peers behind NATs which in turn might enable deanonymization attacks [58, 126]. Another problem is that the decrease in available globally routable IPv4 addresses [85] will in the near future sharply reduce the fraction of hosts that would be able to facilitate NAT traversal.

3.2 Technical Approach

The proposed technique assumes that the client has somehow learned the current external (globally routable) IP address of the server's NAT. This could be due to a previous connection between the two systems, a third party having provided the IP address in a previous exchange or some other out of band method (e.g. email, phone call). Note that we specifically assume that no third party is available at the time when the client attempts to connect to the server behind the NAT, or that the server does not subscribe to any of the third party services which facilitate NAT traversal.

The first goal of the presented NAT traversal method is to communicate the public IP address of a client that wants to connect to the server behind the NAT. After the server is aware of the IP address of the client, it connects to the client (similar to NAT traversal methods that involve a third party).

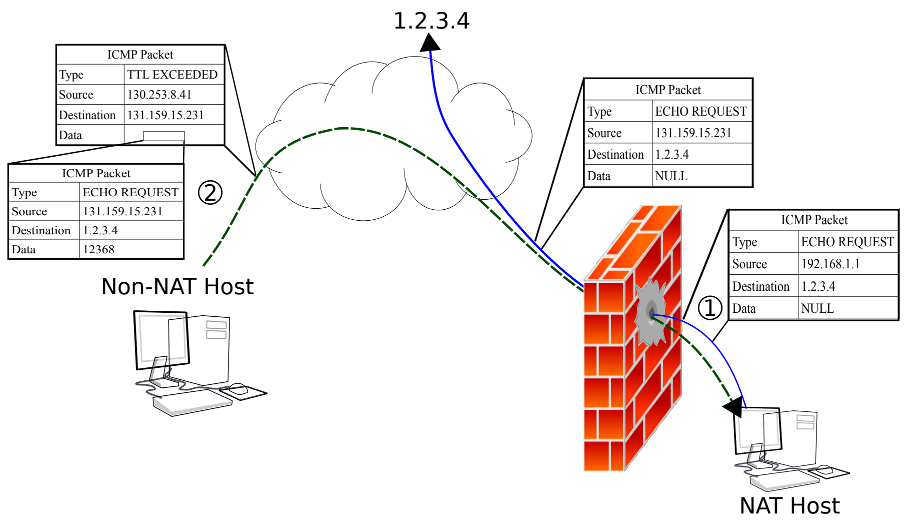


Fig. 3.1: This figure diagrams the process of sending and receiving the fake ICMP messages for the server and client. In step 1, the server sends a fake ICMP request to 1.2.3.4 and in step 2 the client sends the matching reply. Note that this is a fake reply since the client *never* receives the ICMP request sent to 1.2.3.4 by the server. The important information contained in the actual packets is displayed for each step. The blue (solid) line shows the ICMP request path and the dashed (green) line shows the ICMP reply path.

The key idea for enabling the server to learn the client's IP address is for the server to periodically send a message to a fixed, known IP address. The simplest approach uses ICMP ECHO REQUEST messages to an unallocated IP address, such as 1.2.3.4. Since 1.2.3.4 is not allocated, the ICMP REQUEST will will not be routed by routers without a default route; ICMP DESTINATION UNREACHABLE messages that may be created by those routers can just be ignored by the server.

As a result of the messages sent to 1.2.3.4, the NAT will enable routing of replies in response to this request. The connecting client will then fake such a reply. Specifically, the client will transmit an ICMP message indicating `TTL_EXPIRED` (Figure 3.1). Such a message could legitimately be transmitted by any Internet router and the sender address would not be expected to match the server's target IP.

The server listens for (fake) ICMP replies and upon receipt initiates a connection to the sender IP specified in the ICMP reply. If the client is using a globally routable IP address, this is entirely unproblematic and both TCP or UDP can be used to establish a bi-directional connection if the client listens on a pre-agreed port. In cases where there is no pre-agreed port, a port number can in most cases be communicated as part of the payload of the ICMP ECHO RESPONSE, which is typically not checked against the payload of the corresponding ICMP ECHO REQUEST by NAT implementations.

3.2.1 NAT-to-NAT Communication

Further complications arise if both the client and the server are behind NAT. In this case, often the client will be unable to transmit a fake ICMP response to the server due to restrictions imposed by the NAT implementation of the client. One possible idea for circumventing this problem is for the client to send the same message that the server is sending except with TTL 1 to its NAT. If the NAT accepts the packet despite the forged sender IP address it might theoretically generate the desired ICMP response and forward it to the external network. However, in practice we did not find NATs where generating the necessary ICMP message using a TTL of 1 works.

Even if the client is able to transmit the fake ICMP response, the next step; in which both the client and server are aware of the others IP address and now intend to establish a TCP or UDP connection can still be complicated. The reason is that NAT systems can change the source port numbers of outbound messages. Without a third party, both client and server would have to guess matching source and destination port numbers as chosen (possibly at random) by their respective NAT implementations. Depending on the type of the NAT implementations (Full cone, restricted cone, port-restricted, symmetric), finding the correct port may take several messages. Client and server can reduce the total number of messages required by transmitting and listening on multiple ports in this phase.

3.2.2 Using UDP packets instead of ICMP ECHO REQUESTs

A possible alternative to having the sender transmit ICMP ECHO REQUESTs to a fixed, known IP address is having the sender transmit UDP packets to a fixed, known IP address and port. In this case, the client would again forge an ICMP TTL_EXPIRED message, only this time using the UDP format. The main disadvantage of this variation is that the sender has to guess the external UDP sender port number when faking the ICMP response. Since some NAT implementations randomly change those port numbers, the server might have to send UDP packets using multiple sender ports in order to give the client a sufficient chance at guessing correctly.

The main advantage of this technique is that the server no longer needs to send using RAW sockets, which may reduce the privileges required for the server. Note that the server still needs to be able to listen for the ICMP reply, which requires RAW sockets on Linux. In the case of a full-cone NAT, using UDP packets instead of ICMP ECHO REQUESTs also has the advantage of establishing a port mapping which can then be used as an alternative method for contacting the peer.

	Full cone	Restricted Cone	Port-restricted	Symmetric	Overall
Echo-Server	0/4	9/31	37/56	2/3	53/103 (51%)
Echo-Client	1/4	5/34	2/71	2/5	10/123 (8%)
UDP-Server	1/4	26/40	82/91	3/5	121/149 (81%)
ICMP-UDP-Client	1/4	5/34	2/71	2/5	10/123 (9%)
Preserves Ports	0/4	16/43	72/98	6/6	100/162 (62%)
Any Server	1/4	26/40	83/91	3/5	122/149 (82%)
Two-Message Success	0/4	9/31	43/56	2/3	62/103 (60%)

Tab. 3.1: Experimental evaluation. `Echo-Server` lists the number of NAT implementations allowing fake `ICMP TTL_EXPIRED` replies to traverse the NAT device. `Echo-Client` lists the number of NAT implementations that allow clients to transmit fake `ICMP TTL_EXPIRED` messages. `ICMP-UDP-Client/UDP-Server` show results when using UDP packets instead of `ICMP ECHO REQUESTs`. `Preserves Ports` indicates implementations that preserve the sender's local port.`Any server` lists the NATs where either the `ECHO-Server` or the `UDP-Server` work. `Two-Message Success` lists the number of NATs where autonomous NAT traversal succeeds with `Echo-Server` or UDP with port preservation.

Another difference between the two approaches is the possible payload that can be embedded in the response. With ICMP ECHO REQUESTs, the payload can be as big as the packet size permits and is hence only limited by the MTU of the respective physical network. Well-formed ICMP UDP TTL exceeded replies on the other hand can only contain 32 bits of payload: the ICMP TTL_EXCEEDED response contains the first 64 bits of the payload of the original IP packet. In those 64 bits, the 16-bit UDP checksum field and the 16-bit UDP packet length are unverifiable (for NATs that do not track extensive information about outgoing UDP packets) and can hence be used to transmit 32 bits of information to the server (in addition to the sender's IP address). With our approach, either of these payload sizes is enough as we only transmit a port number in addition to the IP address.

3.3 Implementations

This section summarizes the three implementations of the proposed method that we have done so far. All of the presented implementations are freely available from the web pages of the respective projects.

3.3.1 Implementation in NAT-Tester Framework

Our implementation in the NAT-Tester framework was used to gather the data for the data presented in this chapter. It transmits the various packet types (with or without payload) using raw sockets and uses `libpcap` to determine which messages were forwarded by the NAT. The client is currently available for W32 and Linux and must be run with administrator rights. This implementation is useful for researchers interested in exploring the various variations of this and other NAT traversal methods.

3.3.2 Implementation in `pwnat` tool

The `pwnat` tool[1] is a GNU/Linux-only stand-alone implementation of autonomous NAT traversal. After contacting the server behind the NAT, it establishes a channel with TCP-semantics using UDP packets. It supports both client and server behind NAT (if one of the NATs allows the fake ICMP messages to be transmitted). This implementation targets end-users.

3.3.3 Implementation in the GNUnet Framework

Finally, we have created a re-usable implementation of the presented ICMP-based NAT traversal method in GNUnet, GNU's framework for secure peer-to-peer networking [16]. Since the use of ICMP requires the use of non-portable and often privileged system calls, this implementation is split into three main components:

ICMP server
This component is a small program that provides the core ICMP-related functionality for the server. The code periodically generates the ICMP ECHO REQUEST message and also listens for incoming ICMP TTL_EXCEEDED responses. If such a response is received, it simply prints the IP address of the sender to `stdout`. If the ICMP also contains a 16-byte payload, it is interpreted as a port number and also printed.

ICMP client
This component is a small binary which simply sends a single (fake) ICMP message to the IP address specified at the command line. An additional argument can be given which will be interpreted as a port number to be transmitted in the payload of the fake ICMP response message.

Transport plugin
This component implements a GNUnet transport plugin [63] and is thus specific to the GNUnet peer-to-peer framework. Depending on how the peer is configured, it controls ICMP servers or clients and ultimately establishes connections between peers.

Splitting the implementation into these three components has the advantage of minimizing the amount of code that must run with super-user privileges on POSIX systems (by installing the ICMP server and client with

[1] `http://samy.pl/pwnat/`

the SUID bit set). Furthermore, since the ICMP code is platform-specific, this makes it easier to manage this platform-specific part of the code. Finally, this split makes it easy to share the platform-specific but peer-to-peer network agnostic ICMP code so that it can be used with other peer-to-peer applications. This implementation is suitable as a starting point for developers of P2P networks.

3.4 Experimental Results

We have evaluated the proposed autonomous NAT traversal techniques on a large number of NAT implementations using our NAT-Tester framework [123, 124]. The framework consists of a public client that volunteers download and execute. The client then performs various tests against the local NAT implementation and reports the results back to the NAT-Tester server. This enables us to evaluate NAT traversal strategies against a wide range of NAT implementations. Detailed results are made public on the NAT-Tester web page.[2] In this section we will summarize the results based on the data available so far.

Table 3.1 summarizes which fractions of the NAT implementations evaluated so far support the proposed method for autonomous NAT traversal. We distinguish between behavior relevant for using autonomous NAT traversal from the point of view of both clients and servers behind NAT. We consider two cases: the case where the server uses ICMP ECHO REQUESTs and the case where the server transmits UDP packets. We also consider the extend of UDP port randomization which determines how efficient the second stage in the case of NAT-to-NAT communication would be. NAT implementations are categorized into the typical four types (full cone, restricted cone, port-restricted, symmetric) in cases where NAT-Tester is able to determine the type. NAT implementations that do not seem to fall into any of these categories are only included in the total.

The data shows that in virtually all cases NATs forward the faked ICMP messages for UDP (UDP-Server), but only in about half the cases for ICMP ECHO REQUESTs (Echo-Server). Furthermore, a significant majority of all NATs also preserve the source port (when possible), so the additional requirement of guessing the port for faking the ICMP response for a UDP message does not change the overall cost of the approach. Finally, NATs virtually always prevent their clients from transmitting the fake ICMP messages used by our clients (Echo-Client, ICMP-UDP-Client). Based on what we have seen from inspecting NAT configurations directly, the reason seems to be that NAT rules typically only allow ICMP packets for the states “NEW” and “ESTABLISHED” in the state machine [142] — and the fake response falls into neither category.

[2] http://nattest.net.in.tum.de/

3.5 Discussion

The proposed method of autonomous NAT traversal works well in the case of an unrestricted client attempting to initiate a connection to a server behind NAT. Here, in virtually all cases a single ICMP message by the client would be followed by traditional connection reversal [169] which then reliably creates a UDP or TCP connection. In other words, there is no need for third parties to help initiate connections to NATed servers in this case.

On the other hand, if both systems are behind NAT, the proposed method rarely works and a third party is required. Assuming 70% of the peers in a network are behind NAT, this means that roughly 50% of all possible connections can be established using autonomous NAT traversal. However, even in the case where both systems are behind NAT a possible advantage of the proposed method remains; it is easy to create a simple, generic and fully stateless service that receives requests from NATed peers and generates fake ICMP replies to notify the server behind NAT. In this case, the payload of the ICMP reply would need to contain the original IP address (and likely source port number) of the client since the IP header of the faked ICMP response would now contain the IP address of the service.

3.6 Conclusion

Fake replies can enable autonomous NAT traversal in a number of cases. As with most NAT traversal techniques, this approach does not work for all installations. What is unusual about the presented method is that it works extremely well if only one peer is behind NAT and virtually never if both peers are behind NAT. Systems that require high NAT traversal success rates typically implement a number of traversal techniques and the presented approach extends the set of available methods by one that, if applicable, is cheaper and simpler than most of the existing techniques. NAT traversal is inherently unreliable, and limited connectivity may be due to issues other than NAT restrictions such as ad-hoc wireless networks limited by physical range, or social networks limited by relationships. Therefore, NAT traversal by itself is not enough to ensure connectivity depending on the type of underlay network upon which a P2P overlay is built. For these reasons, we have created another method to help provide additional connectivity to peers with limited connectivity, detailed in Chapter 5. This method utilizes onion routing, so the next chapter details an analysis and attack on the dominant onion routing P2P network, Tor. This analysis leads to key choices in our design, discussed further in Chapter 5.

4. A PRACTICAL CONGESTION ATTACK ON TOR USING LONG PATHS

The fish-eye bounded distance vector protocol, which we present in Chapter 5, relies on directly connected peers providing tunneled connections to distant peers. As described in the next chapter, this is very similar to onion routing. This chapter investigates the ability of a specific attack on Tor (which employs onion routing to provide anonymity to users) to perform an asymmetric denial-of-service (DoS) attack. This attack exploits a flaw in the protocol which allows paths of virtually any length to be created. This attack also results in a decrease in anonymity for users of the Tor network. While our distance vector design enables onion routing, we have avoided the flaw that was present in Tor; preventing this specific denial of service attack against our design. The rest of this chapter describes details of the flaw in the design of Tor, the attack, and results from our implementation of the attack on the Tor network. This chapter is based on a paper of the same name published at Usenix Security 2009 [58].

4.1 Introduction

We present an attack which exploits a weakness in Tor's circuit construction protocol to implement an improved variant of Murdoch and Danezis's congestion attack [125, 126]. Tor [45] is an anonymizing peer-to-peer network that provides users with the ability to establish low-latency TCP tunnels, called circuits, through a network of relays provided by the peers in the network. In 2005, Murdoch and Danezis were able to determine the path that messages take through the Tor network by causing congestion in the network and then observing the resulting changes in the traffic patterns.

While Murdoch and Danezis's work popularized the idea proposed in [6] of an adversary perturbing traffic patterns of a low-latency network to deanonymize its users, the original attack no longer works on the modern Tor network. In a network with thousands of relays, too many relays share similar latency characteristics and the amount of congestion that was detectable in 2005 is no longer significant; thus, the traffic of a single normal user does not leave an easily distinguishable signature in the significantly larger volume of data routed by today's Tor network.

We address the original attacks' weaknesses by combining JavaScript injection with a selective and asymmetric denial-of-service (DoS) attack to obtain specific information about the path selected by the victim. As a result, we are able to identify the entire path for a user of today's Tor network. Because our attack magnifies the congestion effects of the original attack, it requires little bandwidth on the part of the attacker. We also provide an improved method for evaluating the statistical significance of the obtained data, based on Tor's message scheduling algorithm. As a result, we are not only able to determine which relays make up the circuit with high probability, we can also quantify the extent to which the attack succeeds. This chapter presents the attack and experimental results obtained from the actual Tor network.

We propose some non-trivial modifications to the current Tor protocol and implementation which would raise the cost of the attack. However, we emphasize that a full defense against our attack is still not known.

Just as Murdoch and Danezis's work applied to other systems such as MorphMix [113] or Tarzan [184], our improved attack and suggested partial defense can also be generalized to other networks using onion routing. Also, in contrast to previously proposed solutions to congestion attacks [74, 99, 101, 113, 129, 133, 164, 184], our proposed modifications do not impact the performance of the anonymizing network.

This attack and analysis led us to careful consideration when using a similar technique to onion routing to provide additional connectivity to peers in very sparse topologies. This technique, presented in Chapter 5, uses encrypted tunnels through the network in a similar manner as Tor. However, we build these tunnels in different ways so as to avoid the DoS attack presented in this chapter.

4.2 Related Work

Chaum's mixes [28] are a commonly used method for achieving anonymity. Multiple encrypted messages are sent to a mix from different sources and each is forwarded by the mix to its respective destination. Combinations of artificial delays, changes in message order, message batching, uniform message formats (after encryption), and chaining of multiple mixes are used to further mask the correspondence between input and output flows in variations of the design [35, 39, 40, 73, 91, 119, 138, 146]. Onion routing [71] is essentially the process of using an initiator-selected chain of low-latency mixes for the transmission of encrypted streams of messages in such a way that each mix only knows the previous and the next mix in the chain, thus providing initiator-anonymity even if some of the mixes are controlled by an adversary.

4.2.1 Tor

Tor [45] is a distributed anonymizing network that uses onion routing to provide its users with anonymity. Most Tor users access the Tor network via a local proxy program such as Privoxy [90] to tunnel the HTTP requests of their browser through the Tor network. The goal is to make it difficult for web servers to ascertain the IP address of the browsing user. Tor provides anonymity by utilizing a large number of distributed volunteer-run relays (or routers). The Tor client software retrieves a list of participating relays, randomly chooses some number of them, and creates a circuit (a chain of relays) through the network. The circuit setup involves establishing a session key with each router in the circuit, so that data sent can be encrypted in multiple layers that are peeled off as the data travels through the network. The client encrypts the data once for each relay, and then sends it to the first relay in the circuit; each relay successively peels off one encryption layer and forwards the traffic to the next link in the chain until it reaches the final node, the exit router of the circuit, which sends the traffic out to the destination on the Internet.

Data that passes through the Tor network is packaged into fixed-sized cells, which are queued upon receipt for processing and forwarding. For each circuit that a Tor router is participating in, the router maintains a separate queue and processes these queues in a round-robin fashion. If a queue for a circuit is empty it is skipped. Other than using this fairness scheme, Tor does not intentionally introduce any latency when forwarding cells.

The threat model assumed by Tor differs from the usual model for anonymity schemes [45]. The traditional threat model is that of a global passive adversary: one that can observe all traffic on the network between any two links. In contrast, Tor assumes a non-global adversary which can only observe some subset of the connections and can control only a subset of Tor nodes. Well-known attack strategies such as blending attacks [163] require more powerful attackers than those permitted by Tor's attacker model. Tor's model is still valuable, as the resulting design achieves a level of anonymity that is sufficient for many users while providing reasonable performance. Unlike the aforementioned strategies, the adversary used in this chapter operates within the limits set by Tor's attacker model. Specifically, our adversary is simply able to run a Tor exit node and access the Tor network with resources similar to those of a normal Tor user.

4.2.2 Attacks on Tor and other Mixes

Many different attacks on low-latency mix networks and other anonymization schemes exist, and a fair number of these are specifically aimed at the Tor network. These attacks can be broadly grouped into three categories: path selection attacks, passive attacks, and active attacks. Path selection attacks attempt to invalidate the assumption that selecting relays at random will usually result in a safe circuit. Passive attacks are those where the adversary in large part simply observes the network in order to reduce the anonymity of users. Active attacks are those where the adversary uses its resources to modify the behavior of the network; we'll focus here on a class of active attacks known as congestion or interference attacks.

4.2.2.1 Path Selection Attacks

Path selection is crucial for the security of Tor users; in order to retain anonymity, the initiator needs to choose a path such that the first and last relay in the circuit won't collude. By selecting relays at random during circuit creation, it could be assumed that the probability of finding at least one non-malicious relay would increase with longer paths. However, this reasoning ignores the possibility that malicious Tor routers might choose only to facilitate connections with other adversary-controlled relays and discard all other connections [22]; thus the initiator either constructs a fully malicious circuit upon randomly selecting a malicious node, or fails that circuit and tries again. This type of attack suggests that longer circuits do not guarantee stronger anonymity.

A variant of this attack called "packet spinning" [133] attempts to force users to select malicious routers by causing legitimate routers to time out. Here the attacker builds circular paths throughout the Tor network and transmits large amounts of data through those paths in order to keep legitimate relays busy. The attacker then runs another set of (malicious) servers which would eventually be selected by users because of the attacker-generated load on all legitimate mixes. The attack is successful if, as a result, the initiator chooses only malicious servers for its circuit, making deanonymization trivial.

4.2.2.2 Passive Attacks

Several passive attacks on mix systems were proposed by Back et al. [6]. The first of these attacks is a "packet counting" attack, where a global passive adversary simply monitors the initiator's output to discover the number of packets sent to the first mix, then observes the first mix to watch for the same number of packets going to some other destination. In this way, a global passive adversary could correlate traffic to a specific user. As described by Levine et al. [101], the main method of defeating such attacks is to pad the links between mixes with cover traffic. This defense is costly and may not solve the problem when faced with an active attacker with significant resources; an adversary with enough bandwidth can deal with cover traffic by using up as much of the allotted traffic between two nodes as possible with adversary-generated traffic [34]. As a result, no remaining bandwidth is available for legitimate cover traffic and the adversary can still deduce the amount of legitimate traffic that is being processed by the mix. This attack (as well as others described in this context) requires the adversary to have significant bandwidth. It should be noted that in contrast, the adversary described by our attack requires only the resources of an average mix operator.

Low-latency anonymity systems are also vulnerable to more active timing analysis variations. The attack presented in [101] is based on an adversary's ability to track specific data through the network by making minor timing modifications to it. The attack assumes that the adversary controls the first and last nodes in the path through the network, with the goal of discovering which destination the initiator is communicating with. The authors discuss both correlating traffic "as is" as well as altering the traffic pattern at the first node in order to make correlation easier at the last node. For this second correlation attack, they describe a packet dropping technique which creates holes in the traffic; these holes then percolate through the network to the last router in the path. The analysis showed that without cover traffic (as employed in Tarzan [68,69]) or defensive dropping [101], it is relatively easy to correlate communications through mix networks. Even with "normal" cover traffic where all packets between nodes look the same, Shmatikov and Wang show that the traffic analysis attacks are still viable [164]. Their proposed solution is to add cover traffic that mimics traffic flows from the initiator's application.

A major limitation of all of the attacks described so far is that while they work well for small networks, they do not scale and may fail to produce reliable results for larger anonymizing networks. For example, Back's active latency measuring attack [6] describes measuring the latencies of circuits and then trying to determine the nodes that were being utilized from the latency of a specific circuit. As the number of nodes grows, this attack becomes more difficult (due to an increased number of possible circuits), especially as more and more circuits have similar latencies.

4.2.2.3 Congestion Attacks

A more powerful relative of the described timing attacks is the clogging or congestion attack. In a clogging attack, the adversary not only monitors the connection between two nodes but also creates paths through other nodes and tries to use all of their available capacity [6]; if one of the nodes in the target path is clogged by the attacker, the observed speed of the victim's connection should change.

In 2005, Murdoch and Danezis described an attack on Tor [126] in which they could reveal all of the routers involved in a Tor circuit. They achieved this result using a combination of a circuit clogging attack and timing analysis. By measuring the load of each node in the network and then subsequently congesting nodes, they were able to discover which nodes were participating in a particular circuit. This result is significant, as it reduces Tor's security during a successful attack to that of a collection of one hop proxies. This particular attack worked well on the fledgling Tor network with approximately fifty nodes; the authors experienced a high success rate and no false positives. However, their clogging attack no longer produces a signal that stands out on the current Tor network with thousands of nodes. Because today's Tor network is more heavily used, circuits are created and destroyed more frequently, so the addition of a single clogging circuit has less impact. Also, the increased traffic transmitted through the routers leads to false positives or false negatives due to normal network fluctuations. We provide details about our attempt to reproduce Murdoch and Danezis's work in Section 4.6.

McLachlan and Hopper [113] propose a similar circuit clogging attack against MorphMix [146], disproving claims made in [184] that MorphMix is invulnerable to such an attack. Because all MorphMix users are *required* to also be mix servers, McLachlan and Hopper achieve a stronger result than Murdoch and Danezis: they can identify not only the circuit, but the user as well.

Hopper et al. [82] build on the original clogging attack idea to construct a network latency attack to guess the location of Tor users. Their attack is two-phase: first use a congestion attack to identify the relays in the circuit, and then build a parallel circuit through those relays to estimate the latency between the victim and the first relay. A key contribution from their work is a more mathematical approach that quantifies the amount of information leaked in bits over time. We also note that without a working congestion attack, the practicality of their overall approach is limited.

4.3 Our Attack

Three features of Tor's design are crucial for our attack. First of all, Tor routers do not introduce any artificial delays when routing requests. As a result, it is easy for an adversary to observe changes in request latency. Second, the addresses of all Tor routers are publicly known and easily obtained from the directory servers. Tor developers are working on extensions to Tor (called bridge nodes [43,44]) that would invalidate this assumption, but this service was not widely used at the time of this writing. Finally, the latest Tor server implementation that was available at the time we concluded our original attacks (Tor version 0.2.0.29-rc) did not restrict users from establishing paths of arbitrary length, meaning that there was no restriction in place to limit constructing long paths through Tor servers.[1] We used a modified

[1] Tor version 0.2.1.3-alpha and later servers restrict path lengths to a maximum of eight because of this work.

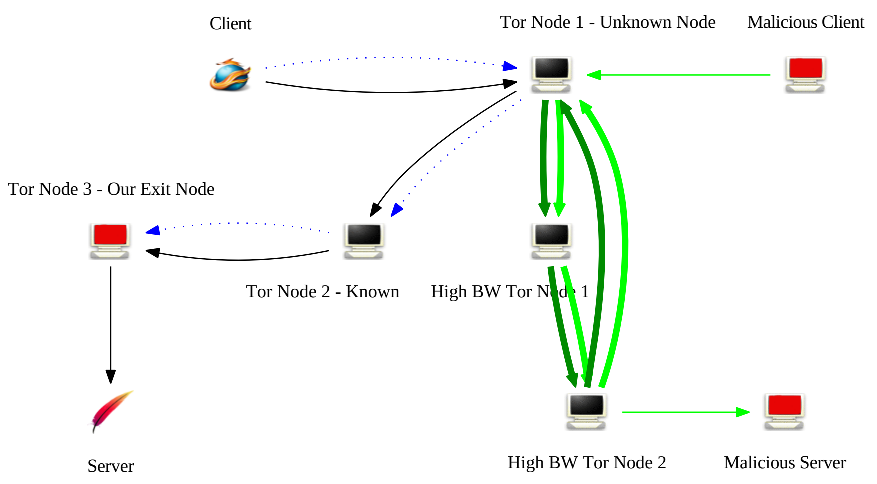


Fig. 4.1: Attack setup. This figure illustrates the normal circuit constructed by the victim to the malicious Tor exit node and the "long" circuit constructed by the attacker to congest the entry (or guard) node used by the victim. The normal thin line from the *client* node to the *server* represents the victim circuit through the Tor network. The unwitting client has chosen the exit server controlled by the adversary, which allows the JavaScript injection. The double thick lines represent the long circular route created by the *malicious client* through the first Tor router chosen by the client. The dotted line shows the path that the JavaScript pings travel.

client version (based on 0.2.0.22-rc) which used a small fixed path length (specifically three) but modified it to use a variable path length specified by our attacker.

Fig. 4.1 illustrates the three main steps of our attack. First, the adversary needs to ensure that the initiator repeatedly performs requests at known intervals. Second, the adversary observes the pattern in arrival times of these requests. Finally, the adversary changes the pattern by selectively performing a novel clogging attack on Tor routers to determine the entry node. We will now describe each of these steps in more detail.

4.3.1 JavaScript Injection

Our attack assumes that the adversary controls an exit node which is used by the victim to access an HTTP server. The attacker uses the Tor exit node to inject a small piece of JavaScript code (shown in Fig. 4.2) into an HTML response. It should be noted that most Tor users do **not** disable JavaScript and that the popular Tor Button plugin [137] and Privoxy [90] also do not disable JavaScript code; doing so would prevent Tor users from accessing

```
<script language="javascript">
 var count,timer,xmlhttp = 0;
 function runonce() {  xmlhttp = new XMLHttpRequest();  }
 function start() {
  xmlhttp.abort();
  xmlhttp = new XMLHttpRequest();
  count++;
  if (timer) clearTimeout(timer);
  timer = setTimeout("start()", 1000);
  myDate = new Date();
  xmlhttp.open("GET", "/reportIn.html?num=" + count + "&
     time="
               + myDate.getTime(),true);
  xmlhttp.send("");
 }
</script>
```

Fig. 4.2: JavaScript code injected by the adversary's exit node. Note that other techniques such as HTML refresh, could also be used to cause the browser to perform periodic requests.

too many web pages. The JavaScript code causes the browser to perform an HTTP request every second, and in response to each request, the adversary uses the exit node to return an empty response, which is thrown away by the browser. Since the JavaScript code may not be able to issue requests precisely every second, it also transmits the local system time (in milliseconds) as part of the request. This allows the adversary to determine the time difference between requests performed by the browser with sufficient precision [2]. While JavaScript is not the only conceivable way for an attacker to cause a browser to transmit data at regular intervals (alternatives include HTTP headers like `refresh` [65] and HTML images [82]), JavaScript provides an easy and generally rather dependable method to generate such a signal.

The adversary then captures the arrival times of the periodic requests performed by the browser. Since the requests are small, an idle Tor network would result in the differences in arrival times being roughly the same as the departure time differences — these are known because they were added by the JavaScript as parameters to the requests. Our experiments suggest that this is often true for the real network, as most routers are not seriously congested most of the time. This is most likely in part due to TCP's congestion control and Tor's built-in load balancing features. Specifically, the variance in latency between the periodic HTTP requests without an active

[2] Clock skew on the systems of the adversary and the victim is usually insignificant for the duration of the attack.

congestion attack is typically in the range of 0–5s.

However, the current Tor network is usually not entirely idle and making the assumption that the victim's circuit is idle is thus not acceptable. Observing congestion on a circuit is not enough to establish that the node under the congestion attack by the adversary is part of the circuit; the circuit may be congested for other reasons. Hence, the adversary needs to also establish a baseline for the congestion of the circuit without an active congestion attack. Establishing measurements for the baseline is done before and after causing congestion in order to ensure that observed changes during the attack are caused by the congestion attack and not due to unrelated changes in network characteristics.

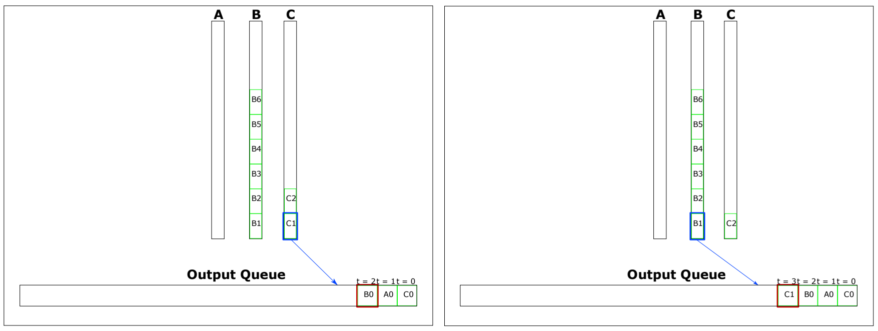
The attacker can repeatedly perform interleaved measurements of both the baseline congestion of the circuit and the congestion of the circuit while attacking a node presumed to be on the circuit. The attacker can continue the measurements until either the victim stops using the circuit or until the mathematical analysis yields a node with a substantially higher deviation from the baseline under congestion compared to all other nodes. Before we can describe details of the mathematical analysis, however, we have to discuss how congestion is expected to impact the latency measurements.

4.3.2 Impact of Congestion on Arrival Times

In order to understand how the congestion attack is expected to impact latency measurements, we first need to take a closer look at how Tor schedules data for routing. Tor makes routing decisions on the level of fixed-size *cells*, each containing 512 bytes of data. Each Tor node routes cells by going round-robin through the list of all circuits, transmitting one packet from each circuit with pending data (see Fig. 4.3a). Usually the number of (active) circuits is small, resulting in little to no delay. If the number of busy circuits is large, messages may start to experience significant delays as the Tor router iterates over the list (see Fig. 4.3b).

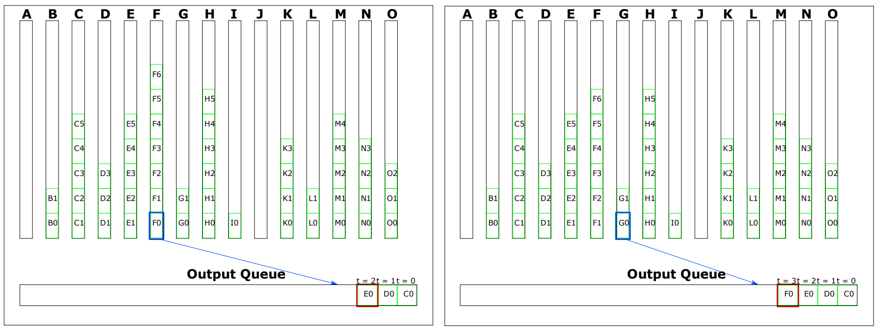
Since the HTTP requests transmitted by the injected JavaScript code are small ($\sim$250 bytes, depending on count and time), more than one request can fit into a single Tor cell. As a result multiple of these requests will be transmitted at the same time if there is congestion at a router. A possible improvement to our attack would be to use a lower level API to send the packets, as the XMLHttpRequest object inserts unnecessary headers into the request/response objects.

We will now characterize the network's behavior under congestion with respect to request arrival times. Assuming that the browser transmits requests at a perfectly steady rate of one request per second, a congested router introducing a delay of (at most) n seconds would cause groups of n HTTP requests to arrive with delays of approximately $0, 1, \ldots, n-1$ seconds respectively: the first cell is delayed by $n - 1$ seconds, the cell arriving a



(a) This example illustrates a Tor router which is handling three circuits at two points in time ($t = 3$ and $t = 4$). Cells are processed one at a time in a round-robin fashion. As the number of circuits increases so does the time to iterate over the queues. The left figure shows the circuit queues and output queue before selection of cell `C1` for output and the right figure shows the queues after queuing `C1` for output. The thicker bottom box of queue C (left) and queue B (right) shows the current position of the round-robin queue iterator. At time $t = 1$ the last cell from queue A was processed leaving the queue A empty so queue A is skipped after processing queue C.

Fig. 4.3



(b) This example illustrates a Tor router under congestion attack handling 15 circuit queues. Note that if a circuit includes a node multiple times, the node assigns the circuit multiple circuit queues. In this example, not all of the circuit queues are busy — this may be because the circuits are not in use or because other routers on the circuit are congested. As in Fig. 4.3a, the left and right figures show the state of the mix before and after queuing a cell, in this case `F0`.

Fig. 4.3: These figures demonstrate the internal queuing in Tor, and why a busy node takes longer to process individual cells.

second later by $n-2$ seconds, and the n-th cell arrives just before the round-robin scheduler processes the circuit and sends all n requests in one batch. This characterization is of course a slight idealization in that it assumes that n is small enough to allow all of the HTTP requests to be grouped into one Tor cell and that there are no other significant fluctuations. Furthermore, it assumes that the amount of congestion caused by the attacker is perfectly steady for the duration of the time measurements, which may not be the case. However, even without these idealizations it is easy to see that the resulting latency histograms would still become "flat" (just not as perfectly regular in terms of arrival patterns) assuming the load caused by the attacker is sufficiently high.

Since we ideally expect delays in message arrival times for a congested circuit to follow a roughly flat distribution between zero and n, it makes sense to compute a histogram of the delays in message arrival times. If the congestion attack is targeting a node on the circuit, we would expect to see a roughly equal number of messages in each interval of the histogram. We will call the shape of the resulting histogram *horizontal*. If the circuit is not congested, we expect to see most messages arrive without significant delay which would place them in the bucket for the lowest latency. We will call the shape of the resulting histogram *vertical*. So for example, in Figure 4.5 the control data are vertical, whereas the attack data are more horizontal.

Note that the clock difference between the victim's system and the adversary as well as the minimal network delay are easily eliminated by normalizing the observed time differences. As a result, the latency histograms should use the increases in latency over the smallest observed latency, not absolute latencies.

4.3.3 Statistical Evaluation

In order to numerically capture congestion at nodes we first measure the node's *baseline* latency, that is, latency without an active congestion attack (at least as far as we know). We then use the observed latencies to create n bins of latency intervals such that each bin contains the same number of data points. Using the χ^2-test we could then determine if the latency pattern at the respective peer has changed "significantly". However, this simplistic test is insufficient. Due to the high level of normal user activity, nodes frequently do change their behavior in terms of latencies, either by becoming congested or by congestion easing due to clients switching to other circuits. For the attacker, congestion easing (the latency histogram getting more vertical) is exactly the opposite of the desired effect. Hence the ordinary χ^2 test should not be applied without modification. What the attacker is looking for is the histogram becoming more horizontal, which for the distribution of the bins means that there are fewer values in the low-latency bins and more values in the high-latency bins. For the medium-latency bins no significant change is expected (and any change there is most likely noise).

Hence we modify our computation of the χ^2 value such that we only include changes in the anticipated direction: for the bins corresponding to the lowest third of the latencies, the square of the difference between expected and observed number of events is only counted in the summation if the number of observed events is lower than expected. For the bins corresponding to the highest third of the latencies, the square of the difference between expected and observed number of events is only counted if the number of observed events is higher than expected. Since changes to the bins in the middle third are most likely noise, those bins are not included in the χ^2 calculation at all (except as a single additional degree of freedom).

Using this method, a single iteration of measuring the baseline and then determining that there was a significant increase in latency (evidenced by a large χ^2-value), only signifies that congestion at the guard for the victim circuit was correlated (in time) with the congestion caused by the attacker. Of course, correlation does not imply causality; in fact, for short (30–60 s) attack runs it frequently happens that the observed χ^2-value is higher for some false-positive node than when attacking the correct guard node. However, such accidental correlations virtually never survive iterated measurements of the latency baseline and χ^2-values under attack.

4.3.4 Congestion Attack

Now we focus on how the attacker controlling the exit node of the circuit will cause significant congestion at nodes that are suspected to be part of the circuit. In general, we will assume that all Tor routers are suspects and that in the simplest case, the attacker will iterate over all known Tor routers with the goal of finding which of these routers is the entry point of the circuit.

For each router X, the attacker constructs a long circuit that repeatedly includes X on the path. Since Tor relays will tear down a circuit that tries to extend to the previous node, we have to use two (or more) other (preferably high-bandwidth) Tor routers before looping back to X. Note that the attacker could choose two different (involuntary) helper nodes in each loop involving X. Since X does not know that the circuit has looped back to X, Tor will treat the long attack circuit as many different circuits when it comes to packet scheduling (Fig. 4.3b).

Once the circuit is sufficiently long (we typically found 24 hops to be effective, but in general this depends on the amount of congestion established during the baseline measurements), the attacker uses the circuit to transmit data. Note that a circuit of length m would allow an attacker with p bandwidth to consume $m \cdot p$ bandwidth on the Tor network, with X routing as much as $\frac{m \cdot p}{3}$ bandwidth. Since X now has to iterate over an additional $\frac{m}{3}$ circuits, this allows the attacker to introduce large delays at this specific router. The main limitation for the attacker here is time. The larger the desired delay d and the smaller the available attacker bandwidth p the longer it will take to construct an attack circuit of sufficient length m: the number of times that the victim node is part of the attack circuit is proportional to the length of the circuit m. In other words, the relationship between p, m and the delay d is $d \sim p \cdot m$.

If the router X is independent of the victim circuit, the measured delays should not change significantly when the attack is running. If X is the entry node, the attacker should observe a delay pattern that matches the power of the attack – resulting in a horizontal latency variance histogram as described in Section 4.3.2. The attacker can vary the strength of the attack (or just switch the long attack circuit between idle and busy a few times) to confirm that the victim's circuit latency changes correlate with the attack. It should be noted that the attacker should be careful to not make the congestion attack too powerful, especially for low-bandwidth targets. In our experiments we sometimes knocked out routers (for a while) by giving them far too much traffic. As a result, instead of receiving requests from the JavaScript code with increasing latencies, the attacker suddenly no longer receives requests at all, which gives no useful data for the statistical evaluation.

4.3.5 Optimizations

The adversary can establish many long circuits to be used for attacks before trying to de-anonymize a particular victim. Since idle circuits would not have any impact on measuring the baseline (or the impact of using another attack circuit), this technique allows an adversary to eliminate the time needed to establish circuits. As users can only be expected to run their browser for a few minutes, eliminating this delay may be important in practice; even users that may use their browser for hours are likely to change between pages (which might cause Tor to change exit nodes) or disable Tor.

In order to further speed up the process, an adversary can try to perform a binary search for X by initially running attacks on half of the routers in the Tor network. With pre-built attack circuits adding an almost unbounded multiplier to the adversary's resources, it is conceivable that a sophisticated attacker could probe a network of size s in $\log_2 s$ rounds of attacks.

In practice, pre-building a single circuit that would cause congestion for half the network is not feasible; the Tor network is not stable enough to sustain circuits that are thousands of hops long. Furthermore, the differences in available bandwidth between the routers complicates the path selection process. In practice, an adversary would most likely pre-build many circuits of moderate size, forgoing some theoretical bandwidth and attack duration reductions for circuits that are more reliable. Furthermore, the adversary may be able to exclude certain Tor routers from the set of candidates for the first hop based on the overall round-trip latency of the victim's circuit. The

Tor network allows the adversary to measure the latency between any two Tor routers [82, 126]; if the overall latency of the victim's circuit is smaller than the latency between the known second router on the path and another router Y, then Y is most likely not a candidate for the entry point.

Finally, the adversary needs to take into consideration that by default, a Tor user switches circuits every 10 minutes. This further limits the window of opportunity for the attacker. However, depending on the browser, the adversary may be able to cause the browser to pipeline HTTP requests which would not allow Tor to switch circuits (since the HTTP session would not end). Tor's circuit switching also has advantages for the adversary: every 10 minutes there is a new chance that the adversary-controlled exit node is chosen by a particular victim. Since users only use a small number of nodes for the first node on a circuit (these nodes are called guard nodes [131]), the adversary has a reasonable chance over time to determine these guard nodes. Compromising one of the guard nodes would then allow full deanonymization of the target user.

4.4 Experimental Results

The results for this chapter were obtained by attacking Tor routers on the real, deployed Tor network (initial measurements were done during the Spring and Summer of 2008; additional data was gathered in Spring 2009 with an insignificantly modified attacker setup; the modifications were needed because our original attack client was too outdated to work with the majority of Tor routers at the time). In order to confirm the accuracy of our experiments and avoid ethical problems, we did not attempt to de-anonymize real users. Instead, we established our own client circuits through the Tor network to our malicious exit node and then confirmed that our statistical analysis was able to determine the entry node used by our own client. Both the entry nodes and the second nodes on the circuits were normal nodes in the Tor network outside of our control.

The various roles associated with the adversary (exit node, malicious circuit client, and malicious circuit webserver) as well as the "de-anonymized" victim were distributed across different machines in order to minimize interference between the attacking systems and the targeted systems. For the measurements we had the simulated victim running a browser requesting and executing the malicious JavaScript code, as well as a machine running the listening server to which the client transmits the "ping" signal approximately every second (Fig. 4.1). The browser always connected to the same unmodified Tor client via Privoxy [90]. The Tor client used the standard configuration except that we configured it to use our malicious exit node for its circuits. The other two nodes in the circuit were chosen at random by Tor. Our malicious exit node participated as a normal Tor router in the Tor network for the duration of the study (approximately six weeks).

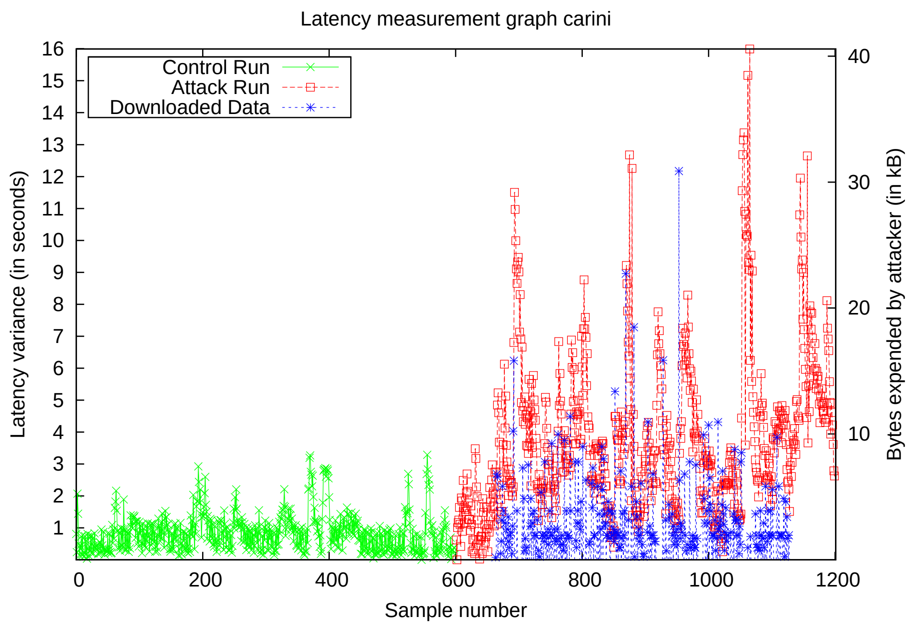
For our tests we did not actually make the exit server inject the JavaScript code; while this is a relatively trivial modification to the Tor code we used a simplified setup with a webserver serving pages with the JavaScript code already present.

The congestion part of the attack requires three components: a simple HTTP server serving an "infinite" stream of random data, a simple HTTP client downloading this stream of data via Tor, and finally a modified Tor client that constructs "long" circuits through those Tor nodes that the attacker would like to congest. Specifically, the modified Tor client allows the attacker to choose two (or more) routers with high bandwidth and a specific target Tor node, and build a long circuit by repeatedly alternating between the target node and the other high bandwidth nodes. The circuit is eventually terminated by connecting from some high-bandwidth exit node to the attacker's HTTP server which serves the "infinite" stream of random data as fast as the network can process it. As a result, the attacker maximizes the utilization of the Tor circuit. Even so, an attacker with significant bandwidth can elect to build multiple circuits in parallel or build shorter circuits and still exhaust the bandwidth resources of the target Tor router.

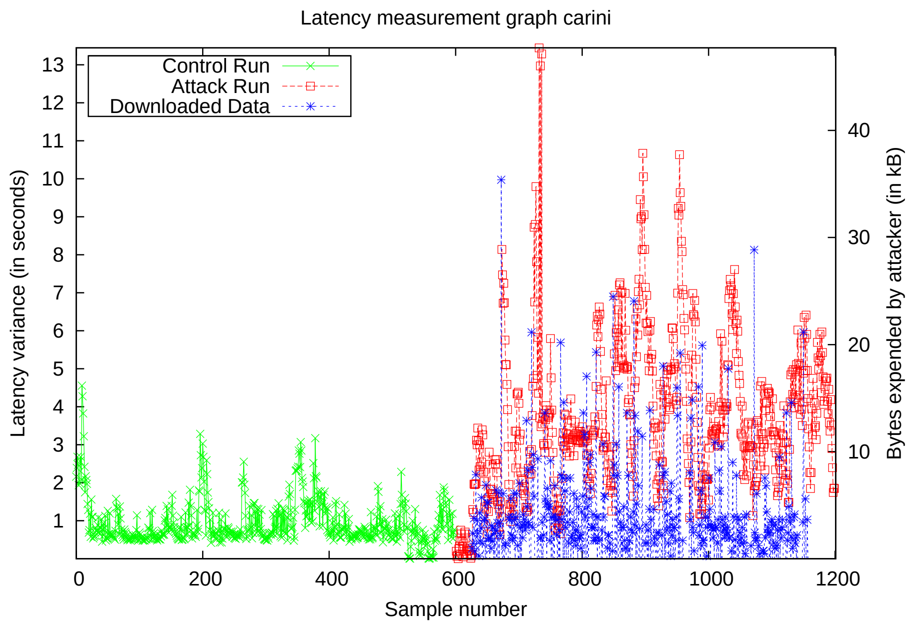
In order to cause congestion, we simply started the malicious client Tor process with the three chosen Tor routers and route length as parameters and then attempted to connect via `libcurl` [37] to the respective malicious server process. The amount of data received was recorded in order to determine bandwidth consumed during the tests. In order to further increase the load on the Tor network the experiments presented actually used two identical attacker setups with a total of six machines duplicating the three machine setup described in the previous paragraph. We found path lengths of 24 (making our attack strength eight times the attacker bandwidth) sufficient to alter latencies. The overall strength of the attack was measured by the sum of the number of bytes routed through the Tor network by both attacker setups. For each trial, we waited to receive six hundred responses from the "victim"; since the browser transmitted requests to Tor at roughly one request per second, a trial typically took approximately ten minutes.

In addition to measuring the variance in packet arrival time while congesting a particular Tor router, each trial also included baseline measurements of the "un-congested" network to discover the normal variance in packet arrival time for a particular circuit. As discussed earlier, these baseline measurements are crucial for determining the significance of the effect that the congestion attack has had on the target circuit.

Figure 4.4 illustrates how running the attack on the first hop of a circuit changes the latency of the received HTTP requests generated by the JavaScript code. The figure uses the same style chosen by Murdoch and Danezis [126], except that an additional line was added to indicate the strength of the attack (as measured by the amount of traffic provided by the adversary). For comparison, the first half of each of the figures shows

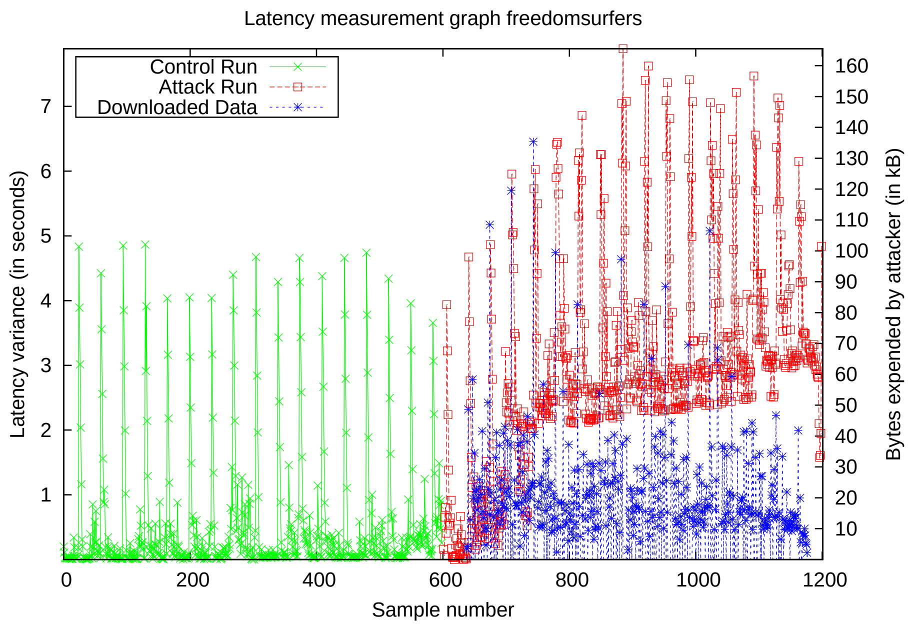

Latency measurement graph carini
Control Run
Attack Run
Downloaded Data
Latency variance (in seconds)
Bytes expended by attacker (in kB)
Sample number
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
40
30
20
10
0
200
400
600
800
1000
1200

(a)

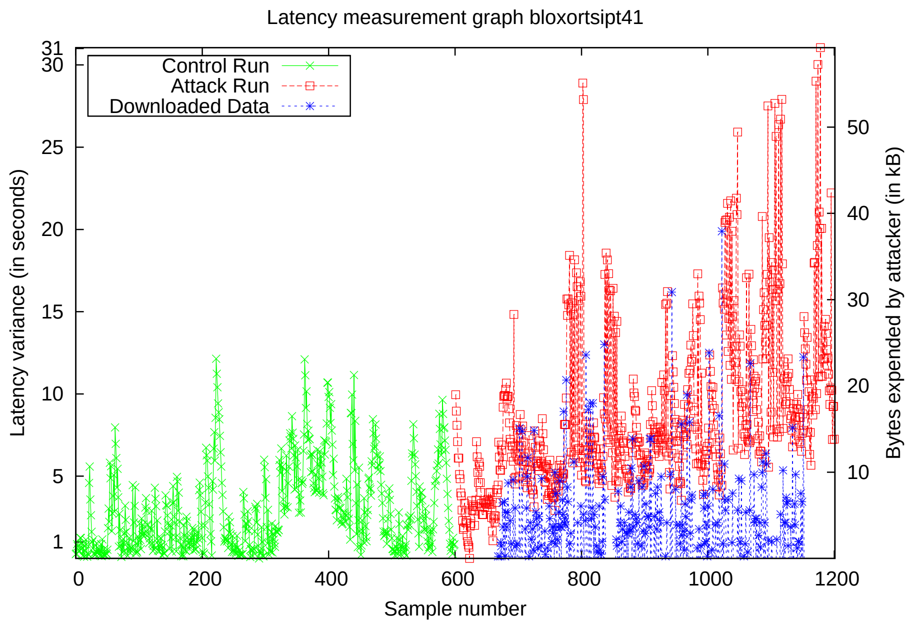

Latency measurement graph carini
Control Run
Attack Run
Downloaded Data
Latency variance (in seconds)
Bytes expended by attacker (in kB)
Sample number
13
12
11
10
9
8
7
6
5
4
3
2
1
40
30
20
10
0
200
400
600
800
1000
1200

(b)

Fig. 4.4



(c)



(d)

Fig. 4.4: Result of of circuit perturbation on latency. The x-axes are sample numbers (one/second), y-axes are latency variance observed on the circuits in seconds. Each attack starts at time 600; the third line shows the amount of data (scaled) that transferred through the attack circuit.

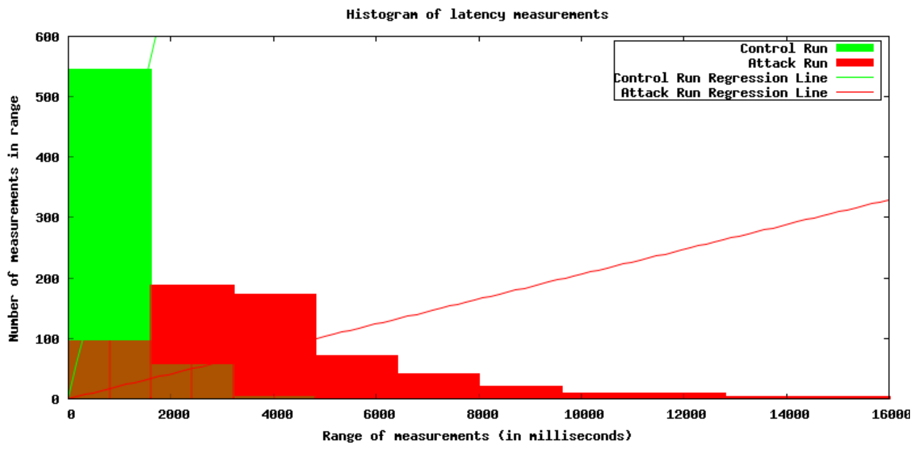
the node latency variance when it is *not* under active congestion attack (or at least not by us).

While the plots in Figure 4.4 visualize the impact of the congestion attack in a simple manner, histograms showing the variance in latency are more suitable to demonstrate the significance of the statistical difference in the traffic patterns. Figure 4.5 shows the artificial delay experienced by requests traveling through the Tor network as observed by the adversary. Since Tor is a low-latency anonymization service, the requests group around a low value for a circuit that is not under attack. As expected, if the entry node is under attack, the delay distribution changes from a steep vertical peak to a mostly horizontal distribution. Figure 4.5 also includes the best-fit linear approximation functions for the latency histograms which we use to characterize how vertical or how horizontal the histogram is as described in Section 4.3.2.

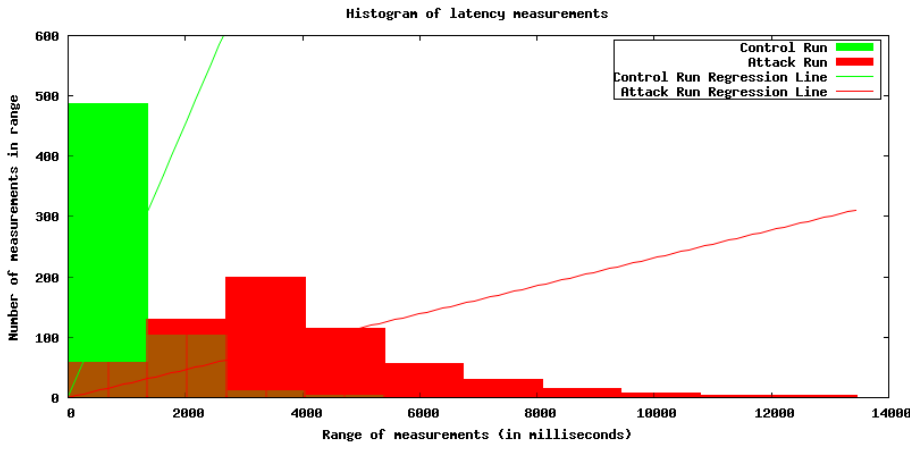
Fig. 4.6 illustrates how the χ^2 values evolve for various nodes over time. Here, we first characterized the baseline congestion for the router for five minutes. Then, the congestion attack was initiated (congesting the various guard nodes). For each attacked node, we used the modified χ^2 summation (from Section 4.3.3) to determine how congested the victim's circuit had become at that time. We computed (cumulative) χ^2 values after 30 s, 60 s, 90 s and so forth. For the χ^2 calculations, we used 60 bins for 300 baseline values; in other words, the time intervals for the bins were chosen so that each bin contained five data points during the five minutes of baseline measurement. The 20 bins in the middle were not included in the summation, resulting in 40 degrees of freedom. As expected, given only 30 s of attack data some "innocent" nodes have higher χ^2 values compared to the entry node (false-positives). However, given more samples, the χ^2 values for those nodes typically drop sharply whereas the χ^2 value when congesting the entry node increases or remains high. Of course, false-positive node's χ^2 values may increase due to network fluctuations over time as well.

Unlucky baseline measurements and shifts in the baseline latency of a router over time can be addressed by iterating between measuring baseline congestion and attack measurements. Fig. 4.7 shows three iterations of first determining the current baseline and then computing χ^2 values under attack. Again the correct entry node exhibits the largest χ^2 values each time after about a minute of gathering latency data under attack.

Given the possibility of false-positives showing up initially when computing χ^2 values, the attacker should target "all" suspected guard nodes for the first few iterations, and then focus his efforts on those nodes that scored highly. Figure 4.8 illustrates this approach. In them, we combine the data from multiple iterations of baseline measurements and χ^2 calculations from attack runs. The attacker determines for each χ^2 value the corresponding confidence interval. These values are frequently large (99.9999% or higher are not uncommon) since Tor routers do frequently experience significant



(a)



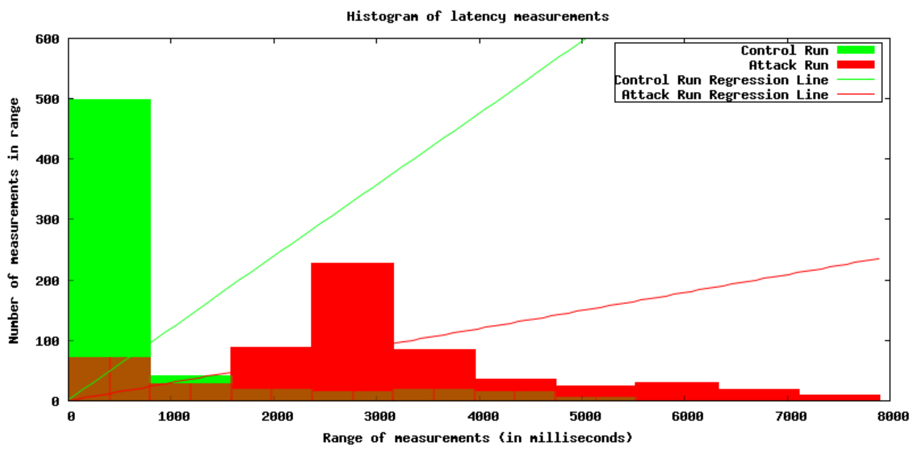
(b)

Fig. 4.5

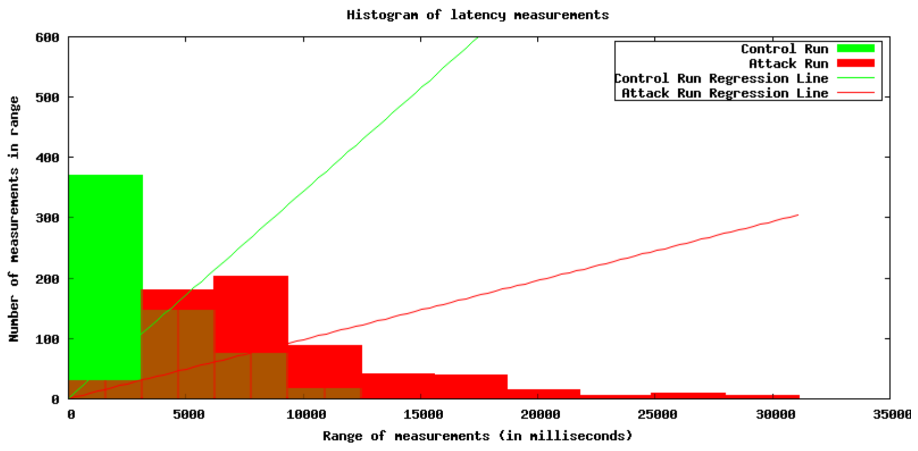
changes in congestion. Given these individual confidence values for each individual iteration, a cumulative score is computed as the product[3] of these values. Figure 4.8 shows the Tor routers with the highest cumulative scores using this metric from trials on two different entry nodes. Note that fewer iterations were performed for routers with low cumulative scores; the router with the highest score (after roughly five iterations) and the most overall iterations is the correctly identified entry node of the circuit in both cases.

Table 4.1 contrasts the product of χ^2 values (as introduced in Sec-

[3] It is conceivable that multiplying χ^2 values may cause false-negatives should a single near-zero χ^2 value for the correct entry node be observed. While we have not encountered this problem in practice, using the mean of χ^2 values would provide a way to avoid this theoretical problem.



(c)



(d)

Fig. 4.5: Results from congestion attack. The x-axes bucket latency variance values, the y-axes are the number of readings received in the range. The hash marked bars represent the unperturbed measurements on a circuit and the plain bars show measurements from the same circuit under attack, which results in a shift to higher latency values. We overlaid linear least squares fit approximations for the baseline and congestion runs.

tion 4.3.3) obtained while attacking the actual first hop with the product while attacking other Tor routers. This shows our attack can be used to distinguish the first hop from other routers when controlling the exit.

Finally, by comparing the highest latency observed during the baseline measurement with the highest latency observed under attack, Table 4.3 provides a simple illustration showing that the congestion attack actually has a significant effect.

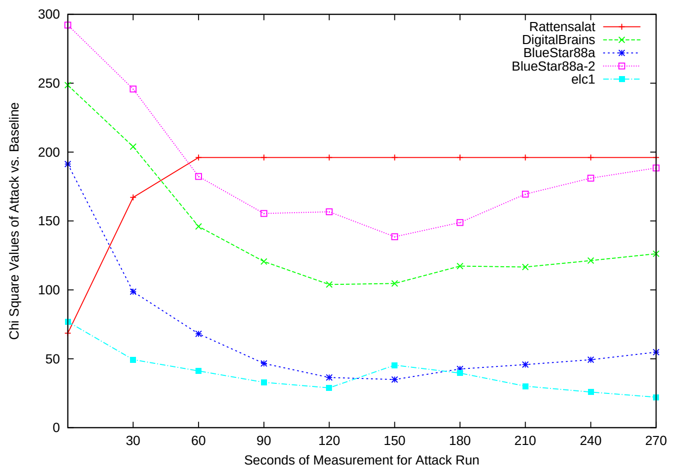


Fig. 4.6: Development of χ^2 values (χ^2 calculation in Section 4.3.3) for performing a congestion attack on the various nodes. χ^2 values computed against a five-minute baseline obtained prior to the congestion attack. The χ^2 value of the correct entry node quickly rises to the top whereas the χ^2 values for all of the other candidates are typically lower after about a minute. A few minutes are typically sufficient to obtain a meaningful χ^2 value.

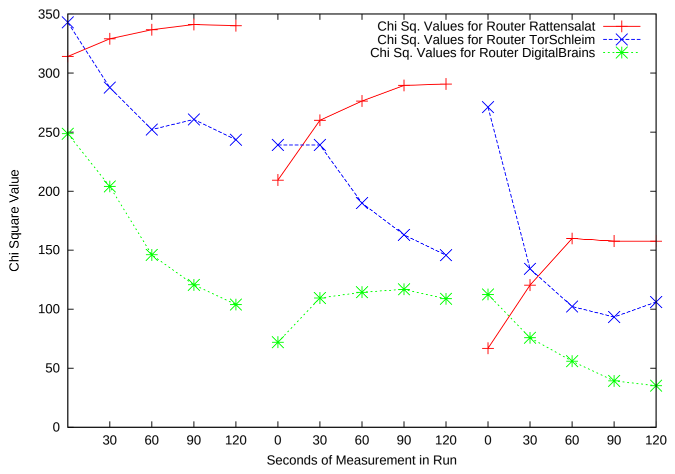
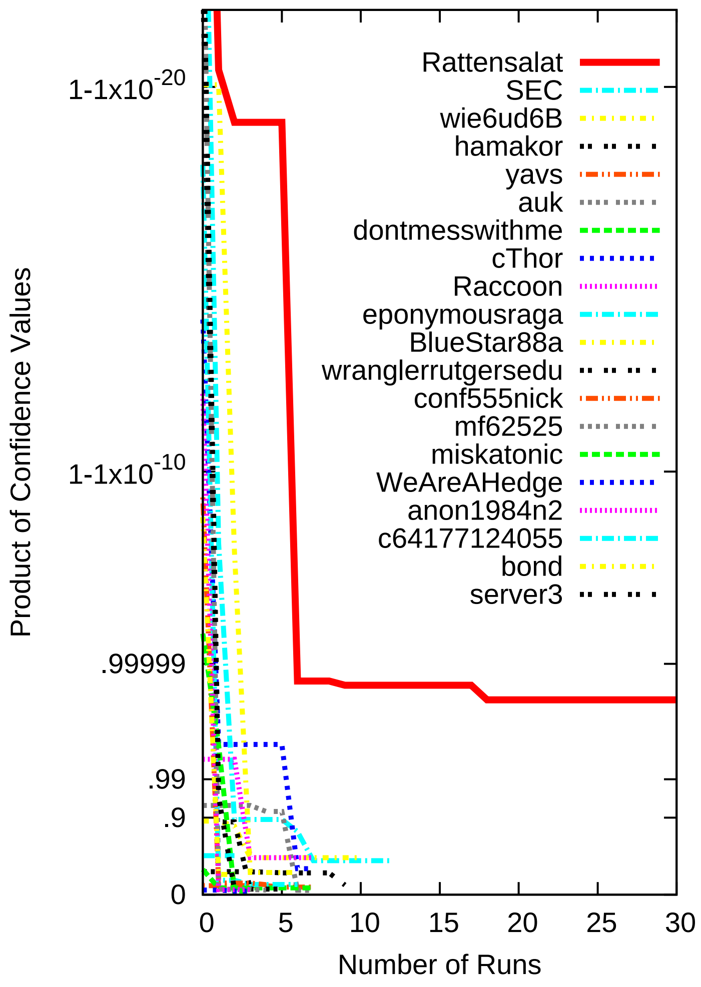
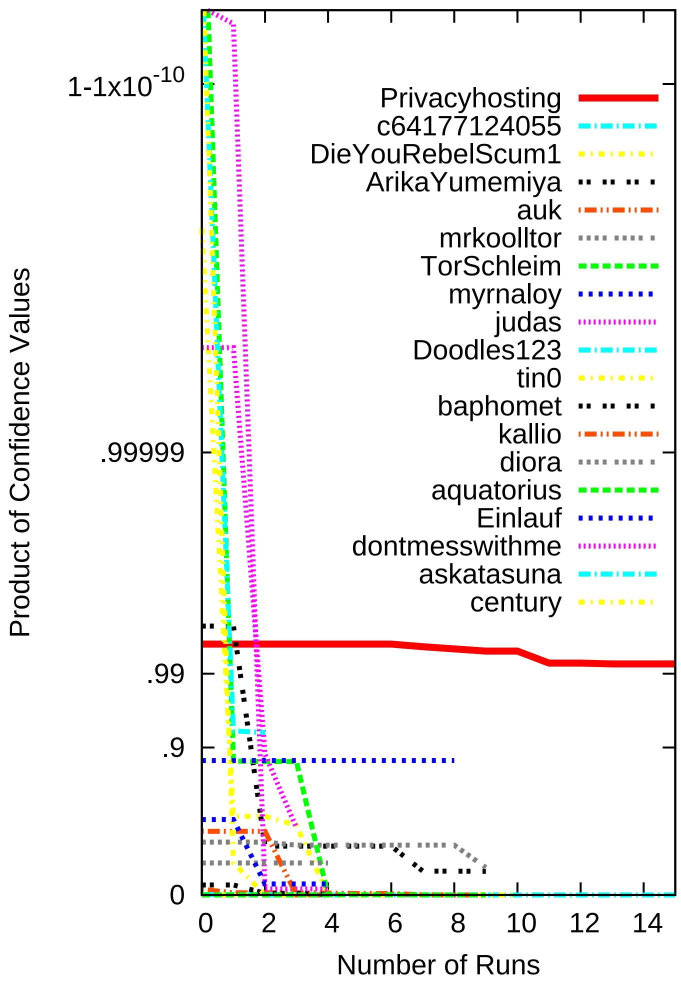


Fig. 4.7: Three sets of cumulative χ^2 computations for three nodes; the actual entry node (`Rattensalat`), a node that initially shows up as a false-positive (`TorSchleim`) and a typical negative (`DigitalBrains`). The χ^2 values (at time 120 s) are consistently the highest for the correct node; false-positives can be ruled out through repeated measurements.



(a)

Fig. 4.8: Plot of the product of χ^2 p-values for the top 20 candidate nodes out of ~250 by run (a run is 300 s baseline vs. 300 s attack) for entry node `Rattensalat`. We expect an attacker to perform more measurements for routers that score high to validate the correct entry node was found. Our measurements demonstrate that the multiplied p-value remains consistently high for the correct entry node. The y-axis is plotted on a log scale from 0 to $1 - 1 \times 10^{-20}$.



(b)

Fig. 4.8: Plot of the product of χ^2 p-values for the top 20 candidate nodes out of ∼200 by run for router `Privacyhosting`. We speculate that the lower maximum value for `Privacyhosting` is due to its higher bandwidth (900 kB/s vs. 231 kB/s for `Rattensalat`).

Router	Πp	r	Peak BW	Configured BW
`Rattensalat`	0.999991	44	231 kB/s	210 kB/s
`c64177124055`	0.903	3	569 kB/s	512 kB/s
`Raccoon`	0.891	8	3337 kB/s	4100 kB/s
`wie6ud6B`	0.890	11	120 kB/s	100 kB/s
`SEC`	0.870	13	4707 kB/s	5120 kB/s
`cThor`	0.789	8	553 kB/s	500 kB/s
`BlueStar88a`	0.734	7	111 kB/s	100 kB/s
`bond`	0.697	3	407 kB/s	384 kB/s
`eponymousraga`	0.458	7	118 kB/s	100 kB/s
`conf555nick`	0.450	5	275 kB/s	200 kB/s

Tab. 4.1: This table lists the top ten (out of 251 total) products of confidence intervals (p-values). r is the number of iterations (and hence the number of factors in Πp) that was performed for the respective router. As expected, the entry node `Rattensalat` achieves the highest score.

We eventually decided upon using the χ^2 distribution test to evaluate our attack because we had some initial difficulty mathematically characterizing how "vertical" or "horizontal" a histogram was in a statistically sound way. Here we detail the first method in which we tried to capture this expected change in distributions using linear least squares best fit approximations. This method correctly captured the difference between baseline and attack trials, but using descriptive statistics to attempt to compare aggregate data proves much more difficult.

We can numerically characterize how vertical or how horizontal a histogram is by computing the angle of a least squares best-fit linear regression function through the origin of the coordinate system and the weighted points of the histogram. For the best-fit, a point representing k measurements in a particular time interval is given weight k. As discussed, based on Tor's cell scheduling algorithm (Fig. 4.3a) and the small message size of the requests generated by the JavaScript code (Fig. 4.2), we would, under ideal circumstances, expect an angle near zero if the node under the congestion attack is part of the circuit and, given suitably large latency intervals, a steep linear approximation function for the baseline histograms (as well as for the case of the congestion attack targeting the wrong node).

The specific numerical angle of the linear approximation function for these histograms is meaningless — the x-axis of the histogram is time and the y-axis is the number of data points; thus, the absolute values cannot even be compared. However, it is possible to establish an expected range for the angles for an uncongested (or vertical) histogram. If the adversary is then able to selectively congest a particular node in the network and obtain a

Router	Πp	r	**Peak BW**	**Configured BW**
`Privacyhosting`	0.9993	17	911 kB/s	5120 kB/s
`askatasuna`	0.994	3	116 kB/s	75 kB/s
`myrnaloy`	0.985	9	4824 kB/s	102400 kB/s
`dontmesswithme`	0.877	4	119 kB/s	100 kB/s
`diora`	0.633	5	1503 kB/s	2048 kB/s
`mrkoolltor`	0.575	10	79 kB/s	64 kB/s
`ArikaYumemiya`	0.520	10	565 kB/s	500 kB/s
`Einlauf`	0.291	5	56 kB/s	50 kB/s
`judas`	0.171	5	311 kB/s	75 kB/s
`baphomet`	0.013	5	208 kB/s	170 kB/s

Tab. 4.2: This table lists the top ten (out of 200 total) products of confidence intervals (p-values). The entry node `Privacyhosting` achieves the highest score.

Router Attacked	**Max Latency Difference**	**Avg. Latency Difference**	**Runs**
`Rattensalat`	70352 ms	25822 ms	41
`Wiia`	46215 ms	470 ms	5
`downtownzion`	39522 ms	2625 ms	9
`dontmesswithme`	37648 ms	166 ms	8
`wie6ud6B`	35058 ms	9628 ms	9
`TorSchleim`	28630 ms	5765 ms	15
`hamakor`	25975 ms	6532 ms	8
`Vault24`	24330 ms	4647 ms	7
`Einlauf`	22626 ms	2017 ms	8
`grsrlfz`	22545 ms	10112 ms	2

Tab. 4.3: This table shows the top ten highest latency differences between the maximum observed measurement in attack runs versus the baseline runs for each router. Unsurprisingly, the difference between the maximum latency observed during the congestion attack and the baseline measurement is significantly higher when attacking the correct first hop compared to attacking other routers. Also included for comparison is the average max latency over all iterations (also higher for the correct first hop), and the number of runs.

latency histogram for the victim's circuit with a linear approximation that has an angle outside of the expected range for vertical histograms, then the congested node is likely to be part of the circuit. Specifically, if the angle of the linear approximation is outside of the $p\%$ confidence interval for un-congested "vertical" histograms, then the probability is $p\%$ that the congested node is part of the circuit. Depending on the stage of the attack, the adversary may preferentially choose to congest a larger set of nodes at the same time. In that case, $p\%$ is the probability that one of the congested nodes is part of the circuit.

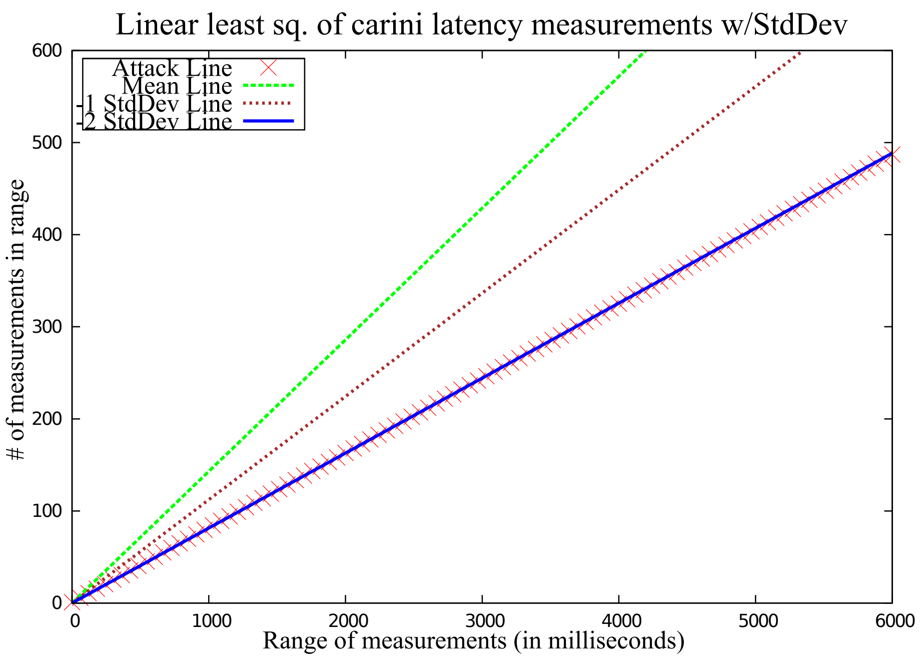
We repeated the baseline measurements to construct an expected range of angles for the approximation function. Figure 4.9 show the average angle of the latency distribution that is expected if the circuit's nodes are not under attack, the expected interval (in standard deviations) and the angle of the same circuit under attack. Remember that if the angle changes significantly, the attacker can be confident that the attacked node is on the circuit.

Table 4.4 lists the confidence levels that we were able to achieve in our experiments for the different circuits and their respective entry-nodes on the Tor network. A high confidence level of c means that either the measurement is in the $1 - c$ fraction of all measurements with a natural significant deviation from the baseline or that the congestion caused by the attack on the node had a significant impact on the latency of the circuit. Table 4.5 contrasts the standard deviation (of the histogram angle from Section 4.3.2) obtained while attacking the first hop with standard deviations observed while attacking other Tor routers. The data shows that our attack can be used to distinguish the first hop from other routers.

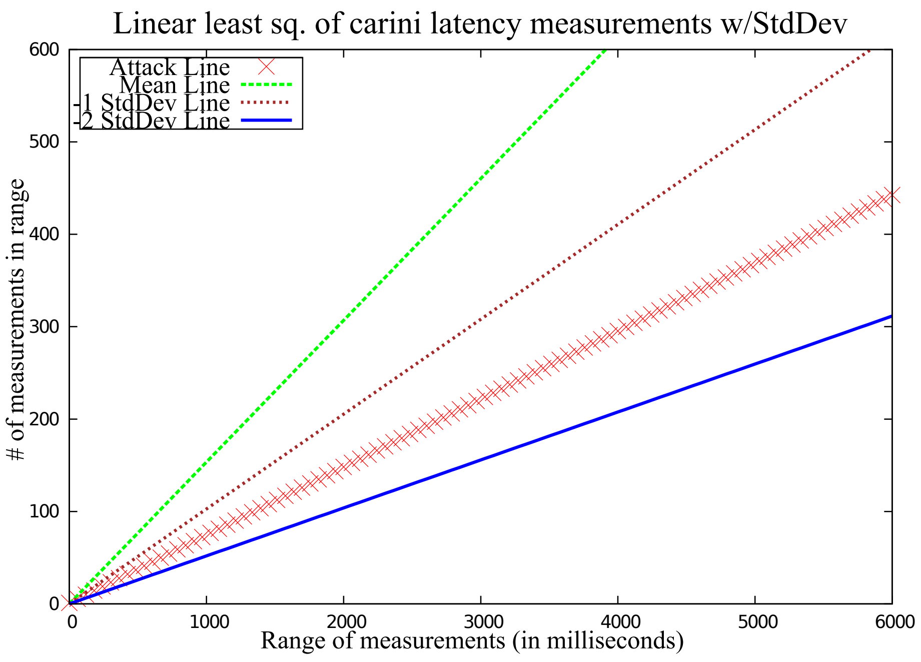
While this method gave us results that were correct, the numbers are not necessarily meaningful in a classic statistical way. For instance, when calculating our range of possible angles, the predicted 3^{rd} standard deviation-out line on either side of the mean line often fell below zero degrees or greater than ninety degrees (i.e. outside of quadrant 1 in a standard Cartesian plane). These values are obviously outside of the range of possible angles, which makes using the percentiles from standard deviations virtually meaningless. However, this method still gave us the most likely candidate, but the resulting numbers lacked the normal statistical meaning which makes the results difficult to understand.

4.5 Proposed Solutions

An immediate workaround that would address the presented attack would be disabling of JavaScript by the end user. However, JavaScript is not the only means by which an attacker could obtain timing information. For example, redirects embedded in the HTML header could also be used (they would, however, be more visible to the end user). Links to images, frames and other features of HTML could also conceivably be used to generate repeated

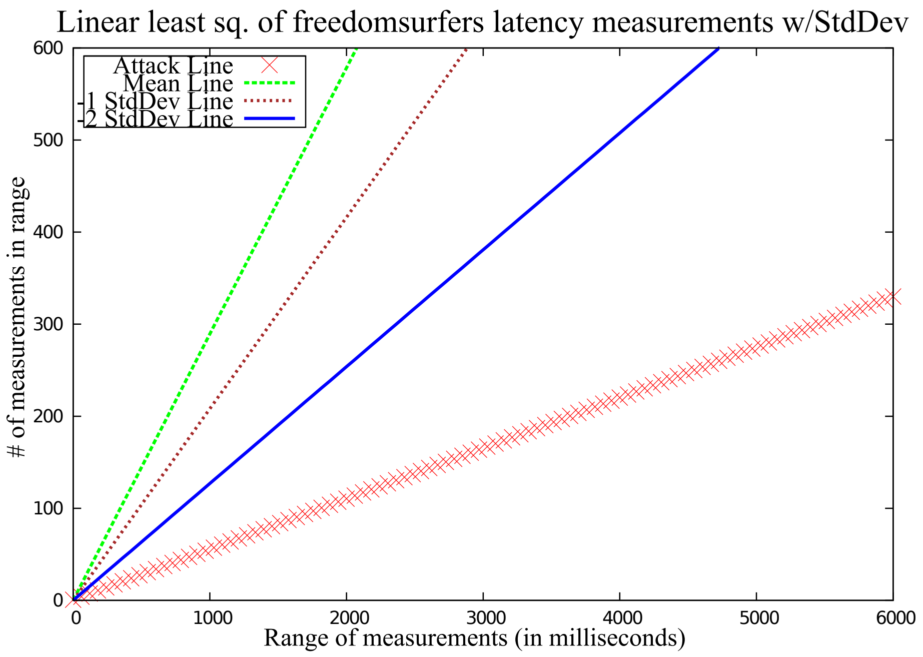

Linear least sq. of carini latency measurements w/StdDev
Attack Line
Mean Line
-1 StdDev Line
-2 StdDev Line
# of measurements in range
Range of measurements (in milliseconds)
0
100
200
300
400
500
600
0
1000
2000
3000
4000
5000
6000

(a)

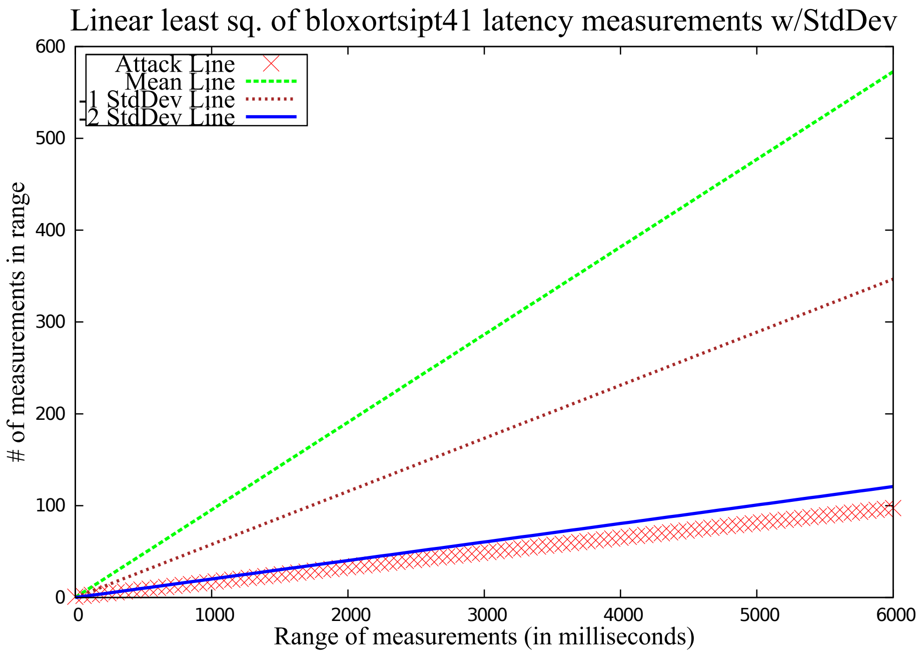

Linear least sq. of carini latency measurements w/StdDev
Attack Line
Mean Line
-1 StdDev Line
-2 StdDev Line
# of measurements in range
Range of measurements (in milliseconds)
0
100
200
300
400
500
600
0
1000
2000
3000
4000
5000
6000

(b)

Fig. 4.9



(c)



(d)

Fig. 4.9: Statistical analysis of the histograms created from the measurements obtained in the Tor network. The subjective results in this light are clear: the attack regression line is typically significantly outside the range of likely values, as objectively shown in Table 4.4.

Router	Peak BW	Configured BW	Avg. Attack Cost	Confidence	Attack Duration
`freedomsurfers`	173.9 kB/s	153.6 kB/s	28.2 kB/s	0.9938	10m
`bloxortsipt41`	54.1 kB/s	51.0 kB/s	4.5 kB/s	0.9826	10m
`carini` (attack 1)	98.2 kB/s	61.4 kB/s	3.0 kB/s	0.9772	10m
`carini` (attack 2)	98.2 kB/s	61.4 kB/s	5.4 kB/s	0.8944	10m
`carini` (combined)	98.2 kB/s	61.4 kB/s	4.2 kB/s	0.9950	20m

Tab. 4.4: This table shows the confidence levels established by our analysis for three circuits and the respective first router of each circuit. The stated confidence that the entry node belongs to the circuit is determined by how far outside the expected range the recorded data under congestion attack was. The table also lists basic properties of the entry node and the duration of the congestion attack. We list the bandwidth of the routers because this is the primary factor that determines how hard it is to congest the node. We obtained geographic information from the `hostip.info` website; the first entry was from Zurich, Switzerland, the second from Florida, USA an the rest from Virginia, USA.

Router	Std. Dev. from Mean	Peak BW	Configured BW
`carini`	2.57	98 kB/s	61 kB/s
`bettyboop`	1.59	2,000 kB/s	102,000 kB/s
`1john2`	1.15	122 kB/s	20 kB/s
`NSAFortMeade`	-0.41	202 kB/s	150 kB/s
`zedz`	-0.44	3,000 kB/s	100,000 kB/s

Tab. 4.5: This table lists the standard deviations from the mean observed in the angle of the linear approximations of latency histograms obtained for a circuit with `carini` as the first hop while congesting various Tor routers (including `carini`). The peak bandwidth is the maximum amount of traffic routed by the respective router in a 10 s interval over the past day. The configured bandwidth is the bandwidth cap specified by the user in the Tor configuration. A negative standard deviation is used to indicate that the latencies were lower than expected (the distribution was more vertical). A high positive standard deviation shows a strong correlation between the attacked router and the first router in the victim circuit.

requests. Disabling all of these features has the disadvantage that the end user's browsing experience would suffer.

A better solution would be to thwart the denial-of-service attack inherent in the Tor protocol. Attackers with limited bandwidth would then no longer be able to significantly impact Tor's performance. Without the ability to selectively increase the latency of a particular Tor router, the resulting timing measurements would most likely give too many false positives. We have extended the Tor protocol to limit the length of a path. The details are described in [42]; we will detail the key points here.

In the modified design, Tor routers now must keep track of how often each circuit has been extended and refuse to route messages that would extend the circuit beyond a given threshold t. This can be done by tagging messages that *may* extend the circuit with a special flag that is not part of the encrypted stream. The easiest way to do this is to introduce a new Tor cell type that is used to flag cells that may extend the circuit. Routers then count the number of messages with the special flag and refuse to route more than a given small number (at the moment, eight) of those messages. Routers that receive a circuit-extension request check that the circuit-extension message is contained in a cell of the appropriate type. Note that these additional checks do not change the performance characteristics of the Tor network. An attacker could still create a long circuit by looping back to an adversary-controlled node every t hops; however, the adversary would then have to provide bandwidth to route every t-th packet; as a result, the bandwidth consumption by the attacker is still bounded by the small constant t instead of the theoretically unbounded path length m.

While this change prevents an attacker from constructing a circuit of arbitrary length, it does not fully prevent the attacker from constructing a path of arbitrary length. The remaining problem is that the attacker could establish a circuit and then from the exit node reconnect to the Tor network again as a client. We could imagine configuring all Tor relays to refuse incoming connections from known exit relays, but even this approach does not entirely solve the problem: the attacker can use any external proxies he likes (e.g. open proxies, unlisted Tor relays, other anonymity networks) to "glue" his circuits together. Assuming external proxies with sufficient aggregate bandwidth are available for gluing, he can build a chain of circuits with arbitrary length. Note that the solution proposed in [133] — limiting circuit construction to trees — does not address this issue; furthermore, it increases overheads and implementation complexity far beyond the change proposed here and (contrary to the claims in [133]) may also have an impact on anonymity, since it requires Tor to fundamentally change the way circuits are constructed. We leave a full solution to this problem as an open research question.

Finally, given that strong adversaries may be able to mount latency altering attacks without Tor's "help", Tor users might consider using a longer path length than the minimalistic default of three. This would involve changes to Tor, as currently the only way for a user to change the default path length would be to edit and recompile the code (probably out of scope for a “normal” user). While the presented attack can be made to work for longer paths, the number of false positives and the time required for a successful path discovery increase significantly with each extra hop. Using a random path length between four and six would furthermore require the adversary to confirm that the first hop was actually found (by determining that none of the other Tor routers could be a predecessor). However, increasing the path length from three to six would significantly increase the latency and bandwidth requirements of the Tor network and might also hurt with respect to other attacks [22].

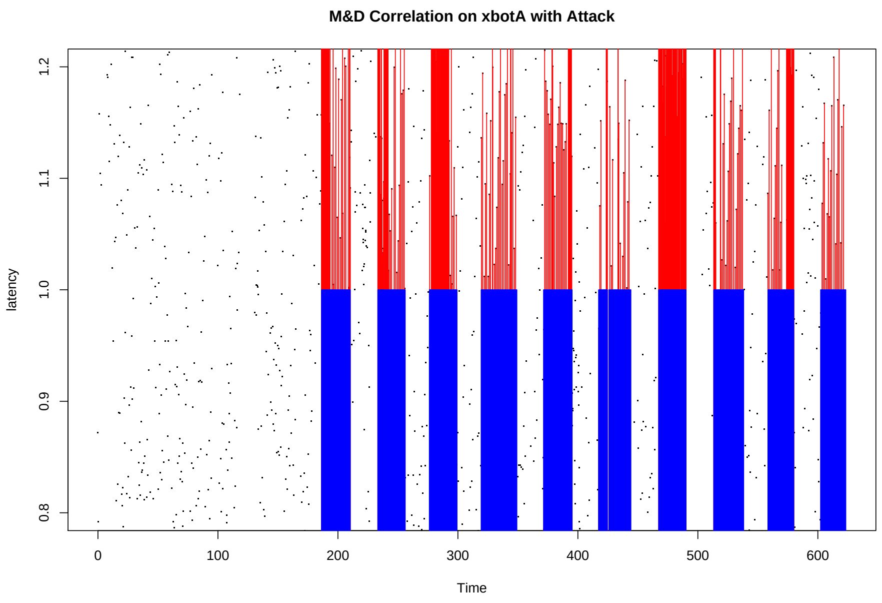
4.6 Low-cost Traffic Analysis Failure Against Modern Tor

We attempted to reproduce Murdoch and Danezis’s work [126] on the Tor network of 2008. Murdoch provided us with their code and statistical analysis framework which performs their congestion attack while measuring the latency of the circuit. Their analysis also determines the average latency and uses normalized latencies as the strength of the signal.

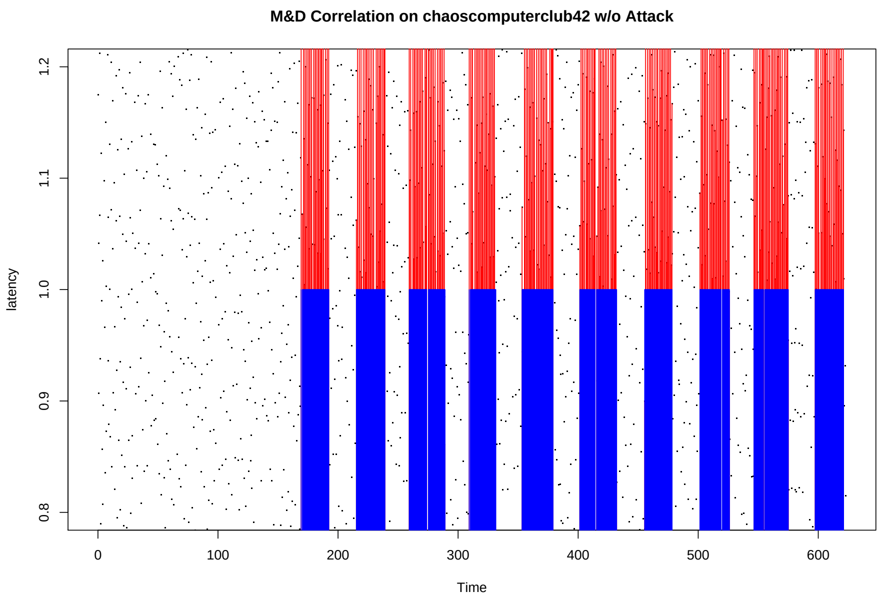
The main difference in terms of how data is obtained between Murdoch and Danezis and the attack presented in Section 4.3 is that Murdoch and Danezis use a circuit constructed by the attacker to measure the latency introduced by the victim circuit whereas our attack uses a circuit constructed by the victim to measure the latency introduced by the attacker.

As described herein, the Murdoch and Danezis style attacker repeatedly switches the congestion attack on and off; a high correlation between the presence of high latency values and the congestion attack being active is used to determine that a particular router is on the circuit. If such a correlation is absent for the correct router, the attack produces false negatives and fails. If a strong correlation is present between high latency values and random time periods (without an active attack) then the attack produces false positives and also fails.

Figure 4.10 shows examples of our attempts at the method used in [126], two with the congestion attack being active and two without. Our experiments reproduced Murdoch and Danezis’s attack setup where the attacker tries to measure the congestion caused by the victim’s circuit. Note that in the graphs on the bottom, the congestion attack was run against a Tor router unrelated to the circuit and thus inactive for the circuit that was measured. Any correlation observed in this case implies that Murdoch and Danezis’s attack produces false positives. The “visual” look of the graphs is the same whether the attack is targeted at that relay or not. Specifically, the graphs on the bottom suggest a similar correlation pattern even when the attack was “off” (or targeting unrelated Tor routers). This is due to the



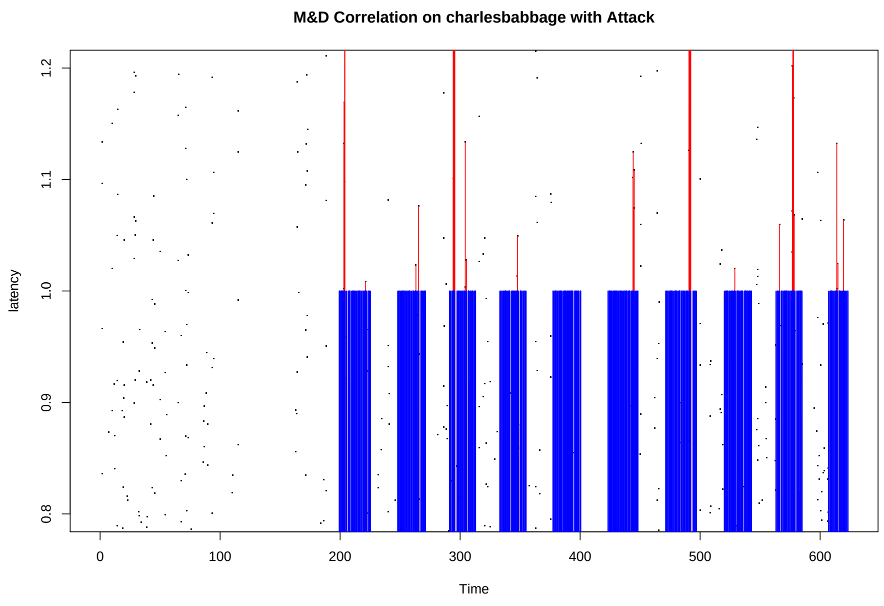
(a)



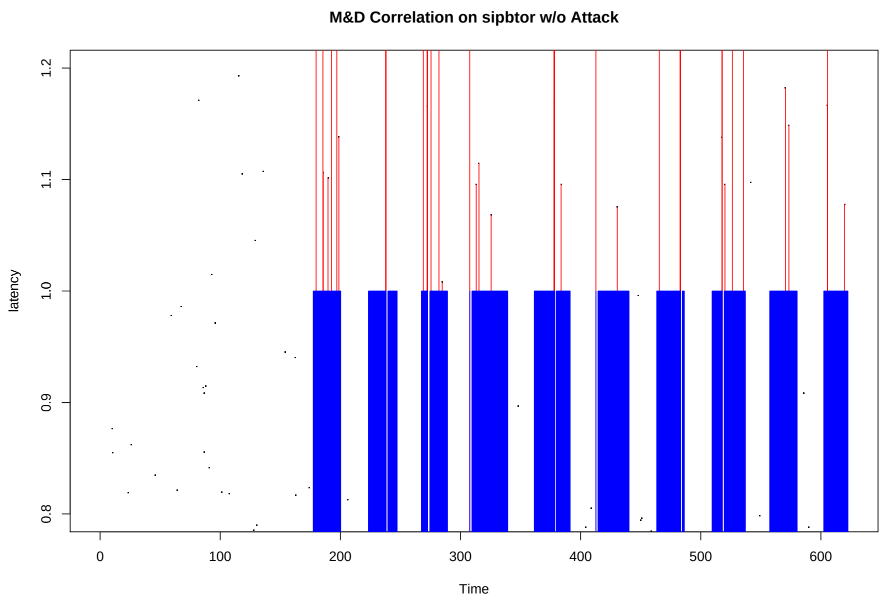
(b)

Fig. 4.10

high volume of traffic on today's Tor network causing baseline congestion which makes their analysis too indiscriminate.



(c)



(d)

Fig. 4.10: Four runs of the method used in [126], two with the congestion attack being active (on top) and two without (on the bottom). The first two are indistinguishable, as are the second two. This shows that the background Tor traffic is too high for their attack to work.

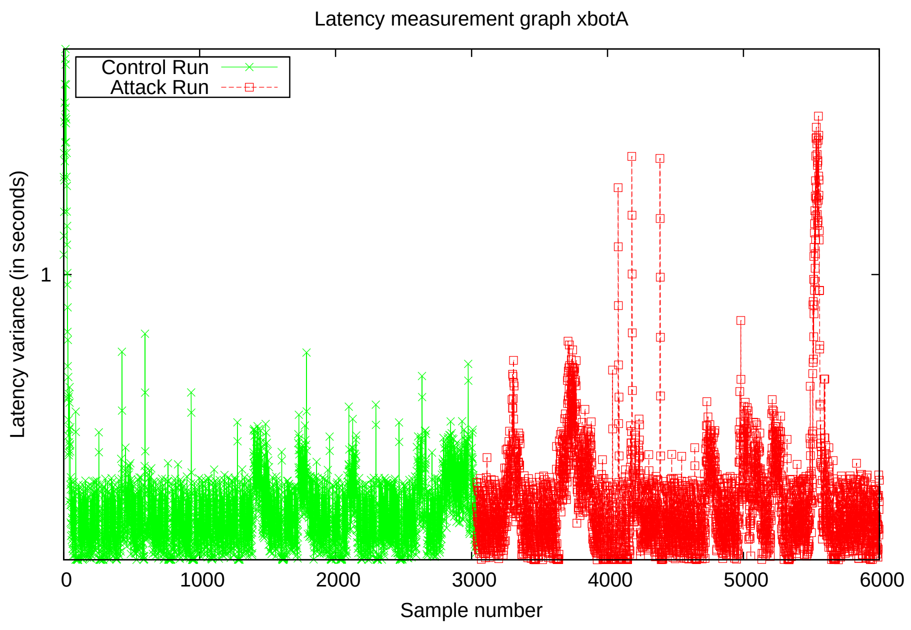
Router	Correlation	Attacked?	Peak BW	Configured BW
`morphiumpherrex`	1.43	Yes	222 kB/s	201 kB/s
`chaoscomputerclub23`	1.34	No	5414 kB/s	5120 kB/s
`humanistischeunion1`	1.18	No	5195 kB/s	6000 kB/s
`mikezhangwithtor`	1.07	No	1848 kB/s	2000 kB/s
`hummingbird`	1.03	No	710 kB/s	600 kB/s
`chaoscomputerclub42`	1.00	Yes	1704 kB/s	5120 kB/s
`degaussYourself`	1.00	No	4013 kB/s	4096 kB/s
`ephemera`	0.91	Yes	445 kB/s	150 kB/s
`fissefjaes`	0.99	Yes	382 kB/s	50 kB/s
`zymurgy`	0.86	Yes	230 kB/s	100 kB/s
`charlesbabbage`	0.53	Yes	2604 kB/s	1300 kB/s

Tab. 4.6: Correlation values calculated using the Murdoch and Danezis's method. False positives and negatives abound.

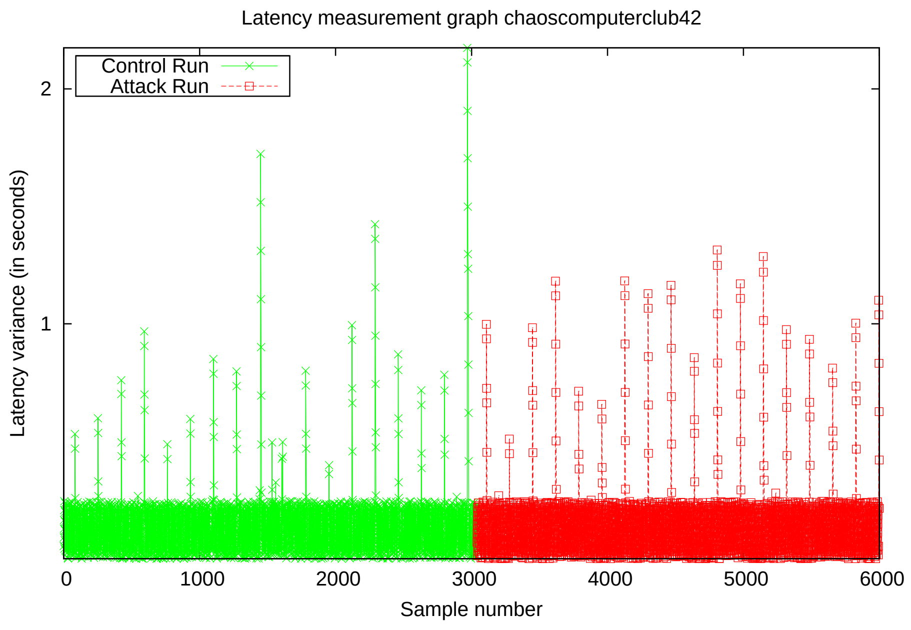
Table 4.6 shows some representative correlation values that were computed using the statistical analysis from [126] when performed on the modern Tor network. Note that the correlation values are high regardless of whether or not the congestion attack was actually performed on the respective router. For Murdoch and Danezis's analysis to work, high correlation values should only appear for the attacked router.

The problem with Murdoch and Danezis's attack and analysis is not primarily with the statistical method; the single-circuit attack itself is simply not generating a sufficiently strong signal on the modern network. Figure 4.11 plots the baseline latencies of Tor routers as well as the latencies of routers subjected to Murdoch and Danezis's congestion attack in the style we used in Figure 4.5. There are hardly any noticeable differences between routers under Murdoch and Danezis's congestion attack and the baseline. Figure 4.12 show the latency histograms for the same data; in contrast to the histograms in Figure 4.5 there is little difference between the histograms for the baseline and the attack.

In conclusion, due to the large amount of traffic on the modern Tor network, Murdoch and Danezis's analysis is unable to differentiate between normal congestion and congestion caused by the attacker; the small amount of congestion caused by their style attack is lost in the noise of the network. As a result, their analysis produces many false positives and false negatives. While these experiments only represent a limited case-study and while Murdoch and Danezis's analysis may still work in some cases, we never got reasonable results on the modern Tor network.

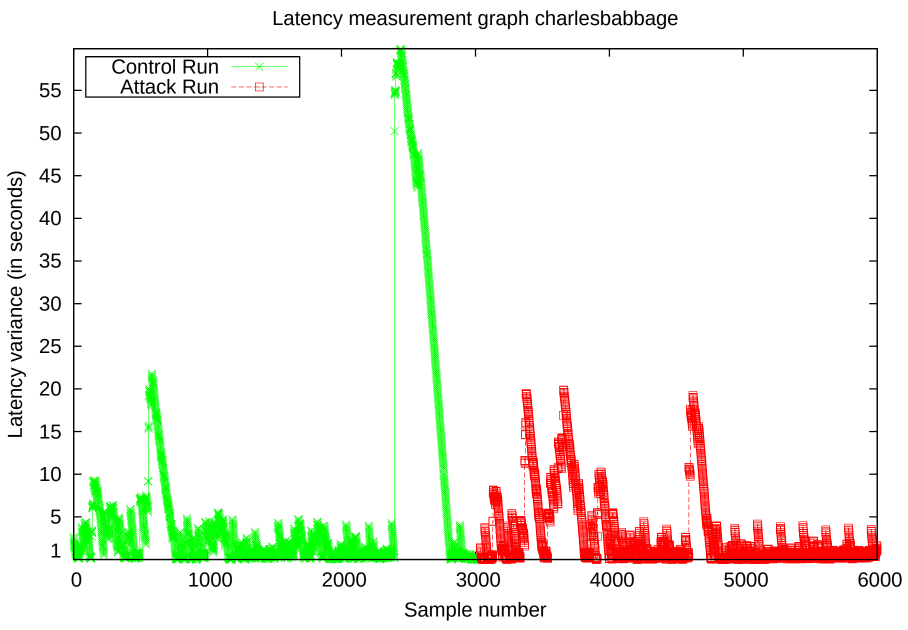

Latency measurement graph xbotA
Control Run
Attack Run
Latency variance (in seconds)
1
0
1000
2000
3000
4000
5000
6000
Sample number

(a)

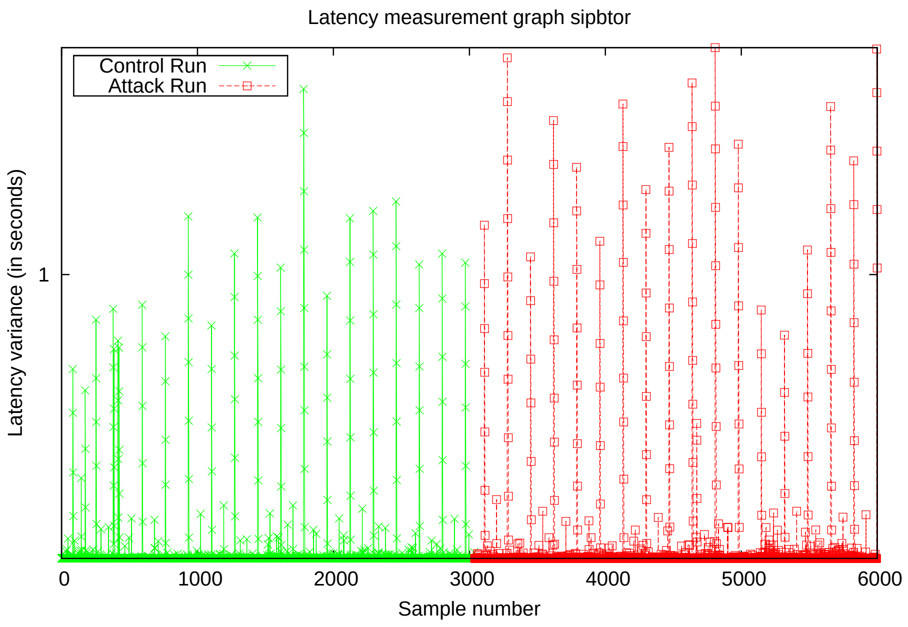

Latency measurement graph chaoscomputerclub42
Control Run
Attack Run
Latency variance (in seconds)
2
1
0
1000
2000
3000
4000
5000
6000
Sample number

(b)

Fig. 4.11

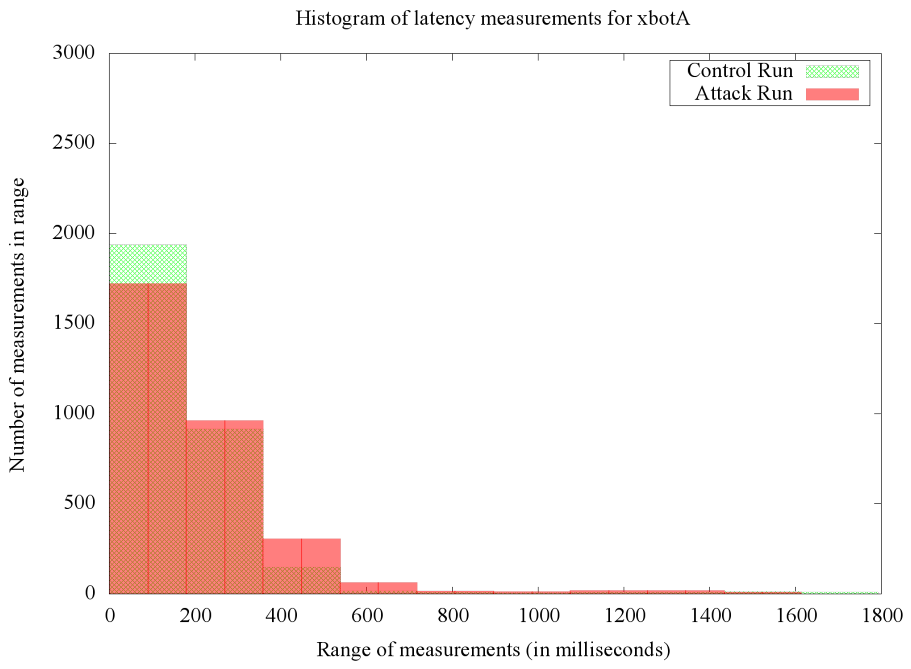


(c)



(d)

Fig. 4.11: Data from Figure 4.10, in the style of Figure 4.4. During the attack phase the congestion circuit is turned on and off as detailed by the original Murdoch and Danezis attack. Latency measurements are virtually identical whether the attack was present or not.


Histogram of latency measurements for xbotA
Control Run
Attack Run
Number of measurements in range
Range of measurements (in milliseconds)
0
500
1000
1500
2000
2500
3000
0
200
400
600
800
1000
1200
1400
1600
1800

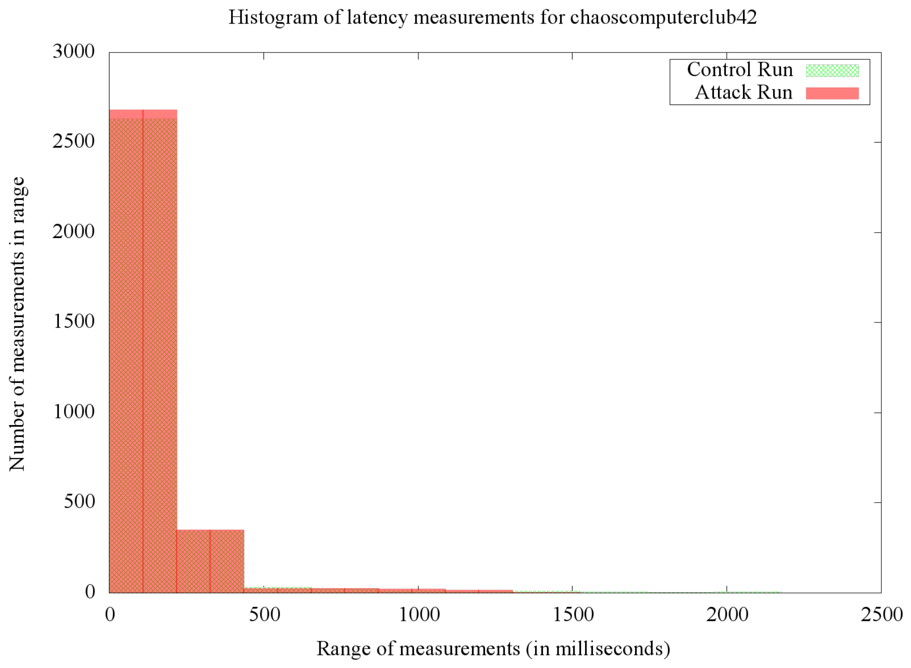
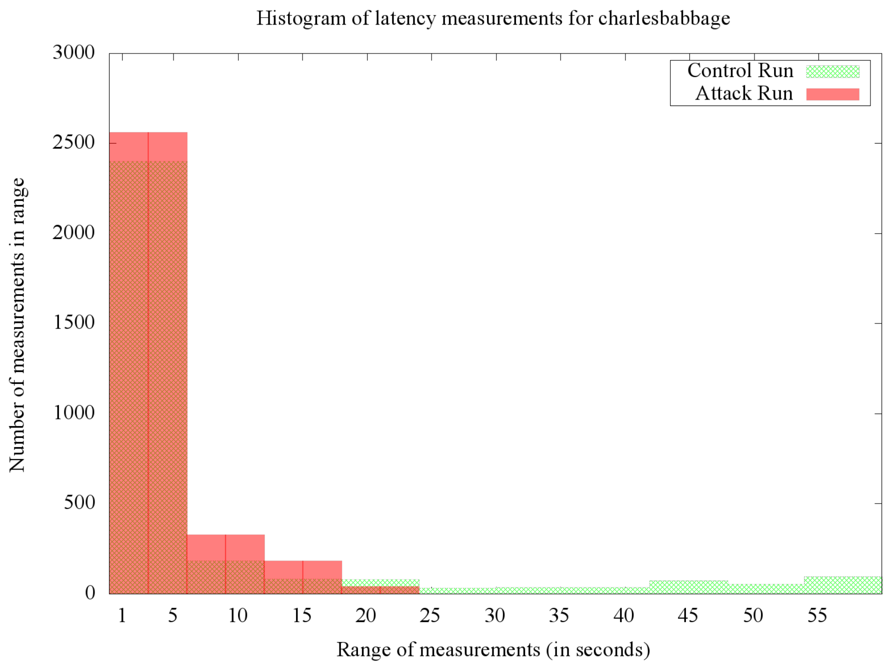

Histogram of latency measurements for chaoscomputerclub42
Control Run
Attack Run
Number of measurements in range
Range of measurements (in milliseconds)
0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500

Fig. 4.12



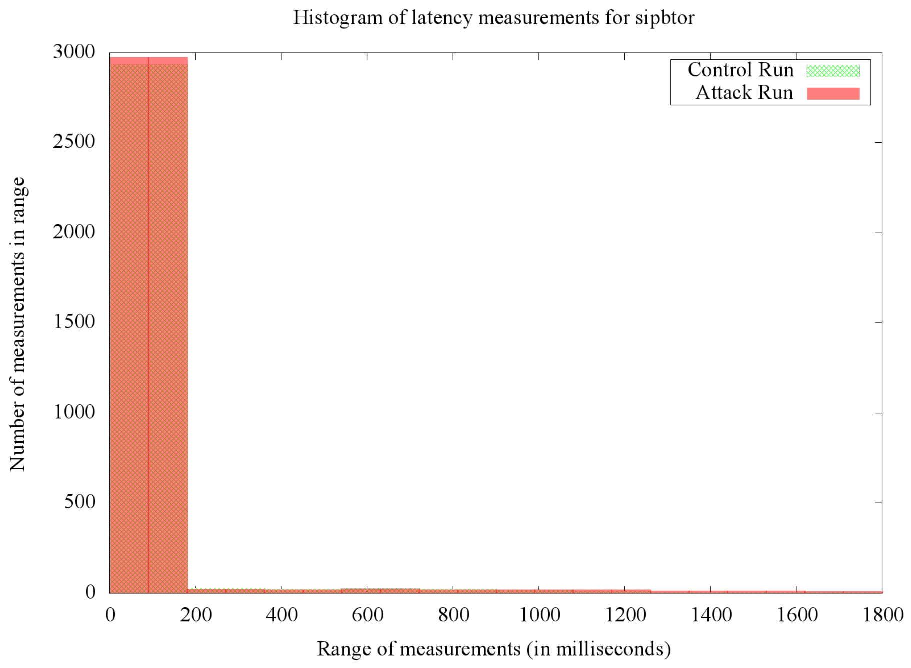


Fig. 4.12: Once more we show the same data for comparison as shown in Figure 4.10, this time in the histogram style we use in Figure 4.5. The overlap between the control run and the attack run is difficult to see due to the similarity of latency distributions.

4.7 Conclusion

The possibility of constructing circuits of arbitrary length was previously seen as a minor problem that could lead to a DoS attack on Tor. This work shows that the problem is more serious, in that an adversary could use such circuits to improve methods for determining the path that packets take through the Tor network. Furthermore, Tor's minimalistic default choice to use circuits of length three is questionable, given that an adversary controlling an exit node would only need to recover a tiny amount of information to learn the entire circuit. We have made some minimal changes to the Tor protocol that make it more difficult (but not impossible) for an adversary to construct long circuits.

The impact of this work on our design is explained in the next chapter. This attack and analysis of Tor led us to be more cautious in our distance vector transport protocol which is designed to give better connectivity to peers using the GNUnet P2P networking framework. This type of transport has similarities to Tor and onion routing, as data is multiply encrypted and routed indirectly to a destination through other "relay" peers. We specifically made design choices that give us protection against peers building routes of arbitrary length through GNUnet using this transport, and thus prevent the possible DoS attack which is inherent in the original Tor design.

5. FISH-EYE BOUNDED DISTANCE VECTOR PROTOCOL

For efficient routing in a DHT, it is quite common for each peer to have at least $O(\log n)$ other peers in its routing table [30,107,112,143,173] (ideally selectively added from the set of all n total peers). However, sparse topologies such as a 2d-grid or wireless mesh networks, or any topology where direct connections can not be made to arbitrary peers, generally do not provide the *specific* $O(\log n)$ connections which are required for efficient DHT routing. For this reason, we have implemented a distance vector routing algorithm which provides connectivity to peers within a certain number of hops distant by relaying data between directly connected peers. Following [10, 136], we call this a "fish-eye bounded distance vector" (FBDV) routing algorithm, where nodes gain local knowledge of the network out to some number of hops (the fish-eye bound) beyond their direct connections. This is achieved by peers gossiping about their nearby neighbors to each other (in the same way that distance vector routing protocols do). This both establishes shortest path routes between close nodes (with relation to hops) as well as providing a greater number of peers for higher level protocols. With the added peers, our DHT routing algorithm can perform more efficiently than it could otherwise in topologies where connectivity is restricted.

5.1 Fish-eye and Zone Routing Protocols

The Bellman-Ford [10] algorithm is a distance vector routing algorithm able to find shortest paths between nodes in a graph iteratively by periodically exchanging cost information. While this algorithm works well for small graphs (where the number of updates and nodes is relatively small) it is problematic when there are a large number of nodes and edges. Specifically, this is because all nodes must continually send updates about all other nodes in the graph until the distances converge, which can be quite costly and time consuming.

The Fish-eye routing design, originally proposed in [136], is a method for maintaining routing state in an efficient manner. Because it is a link state design, every node has knowledge of the global topology of the network. State is updated by each node in the network periodically sending link state information (such as cost) to each other node. The "closer" a node a to another node b, the more often a is informed of the link information of b's neighbors. Essentially, this reduces the communication cost of updating state information because closer nodes are communicated with more often. The clear advantage of this kind of design is that "close" nodes are kept more up to date about a neighbor than "far" nodes which makes sense as long as closer nodes are communicated with more often than more distant nodes.

Our Fish-eye Distance Vector route maintenance is also quite similar to the proactive routing of the Zone Routing Protocol (ZRP) [9]. ZRP combines two routing protocols; traditional flooding distance vector state maintenance is used for a k-hop out neighborhood of nodes and a less expensive (in terms of routing table size) reactive routing protocol is employed for sending messages out of this local neighborhood. We utilize the idea of having a local k-distance neighborhood for our design, but leave routing outside the neighborhood to our higher level (DHT) routing protocol.

The main idea for our combination Fish-eye/ZRP design is that nodes are not aware of global state, but communicate topology information to nodes within a certain maximum "distance" (where distance can be number of hops, link cost, or a combination of the two) to establish a neighborhood of peers. We reduce the cost of maintenance further than [136], as topology information is only sent when new peers are connected or disconnected.

Using the FBDV algorithm provides the benefits of a local distance vector algorithm in a k-size neighborhood while reducing the number of messages sent over costly/distant links. This gives better connectivity for higher level applications, such as our DHT routing algorithm. If sufficient connectivity exists in the network, FBDV is simply not used.

5.2 Implementation

GNUnet is a P2P networking framework which provides connectivity to peers and cryptographic key exchange for encrypted communication between those peers. GNUnet employs a plugin-based transport service, so that multiple types of transports can be used and the best one can be selected based on user specified criteria. For instance, there are UDP and TCP plugins which can both be used at the same time; two peers can be connected via either, both or neither of them. The transport service can choose which of any type of loaded plugins to use for communication depending on measurements i.e., latency, or configuration options such as allowed bandwidth, etc. The FBDV protocol in GNUnet is implemented as a transport plugin, thus it can be used transparently by other P2P applications in the GNUnet system for communication.

GNUnet separates the functionality of a single peer into a number of operating system processes for purposes of fault isolation. This process separation is shown in Figure 5.1, as an example of a simple peer running only four services. Due to this process separation, the distance vector implemen-

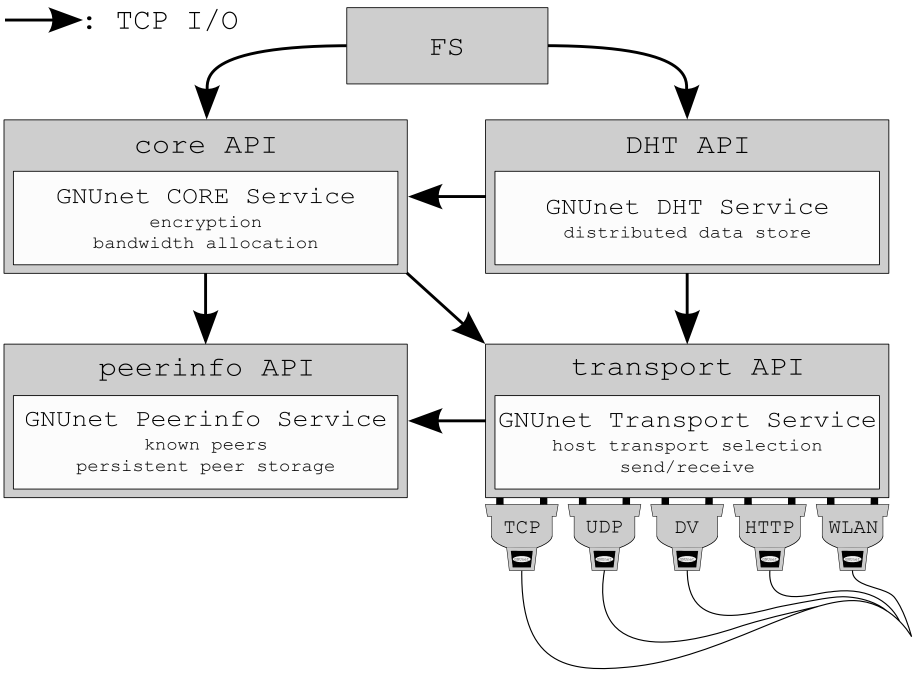


Fig. 5.1: The high level view of a single GNUnet peer. Each box represents a process; directed edges show data flow between these processes. This "peer" is made up of four service processes (core, DHT, peerinfo, transport) and a single user process (FS). The "plugs" under the transport service represent multiple transport plugins enabling low-level communication with other peers.

tation is split into a service which handles communication with the GNUnet core service (providing overlay connections to other peers), and a transport service plugin (providing underlay communication to other peers). The implementation includes the definition of numerous message formats for gossip and data messages which are passed between a single peer's services and also between peers.

Plugins in GNUnet provide addresses and connection related information to the transport service for each peer that the plugin is connected to. This allows the transport service to choose the best plugin and address per peer, which means multiple addresses can exist for a peer via the same plugin as well. This reduces the requirements of the transport plugins; the transport service does the difficult work of selecting the address and plugin for a connection to a peer, and the plugin simply has to send the information. However, the distance vector transport uses connections to other peers, so it requires communication with the GNUnet core service (which manages encrypted connections to other peers, and handles de-multiplexing of messages).

The distance vector service in itself is simple, as its only job is to maintain a table of reachable peers, their associated distances (currently only based on number of hops) and via which directly connected peer the message should be sent. The other job of the service is to encapsulate messages for distant peers in a special distance vector message format and send that to the directly connected peer which the distant peer is reachable from. The distance vector service handles distance vector gossip messages, distance vector data messages, connect and disconnect messages from the core service and provides an API for sending and receiving messages which is used by the distance vector plugin.

One problem with distance vector protocols is preventing looping of messages. Clearly, if a peer c can be reached from a via peer b, then a valid path from a to c could be $a \rightarrow b \rightarrow a \rightarrow b \rightarrow c$. This path has a loop, and is therefore not the most efficient (or helpful) route to choose. To prevent such loops in our distance vector implementation, we use the split horizon method [20] which guards against advertising routes that can create short loops. Also, the count to infinity problem which can also occur is mitigated by our fish-eye bound which reduces the maximum path length allowed in FBDV.

5.3 Distance Vector Service

The distance vector service handles notifications of connect and disconnect events from the peer's core service. Upon notification of a new peer connection, the service creates and sends appropriate distance vector gossip messages. Upon receipt of a distance vector gossip message, the recipient performs checks and subsequently adds or updates the peer entry in the distance vector routing table. When a peer disconnects, the distance vector service propagates disconnect messages to each directly connected peer. These peers remove any entries which are associated with the disconnected peer, and further propagate the disconnect messages if they have further gossiped information about the peer.

5.4 Message Example

The multi-process architecture means sending messages via distance vector requires multiple steps, shown in Figure 5.2. This figure shows these step for sending a message from a peer A to a peer C via peer B. First, an application at peer A notifies the core service that it has a message to send to peer C. Core (A) encrypts the message for C, passing the result to transport (A). Transport (A) calls the distance vector plugin, which in turn triggers a message send request in the distance vector service. The distance vector service encapsulates the data message and passes it to core (A), which encrypts the message for B and hands it to transport (A).

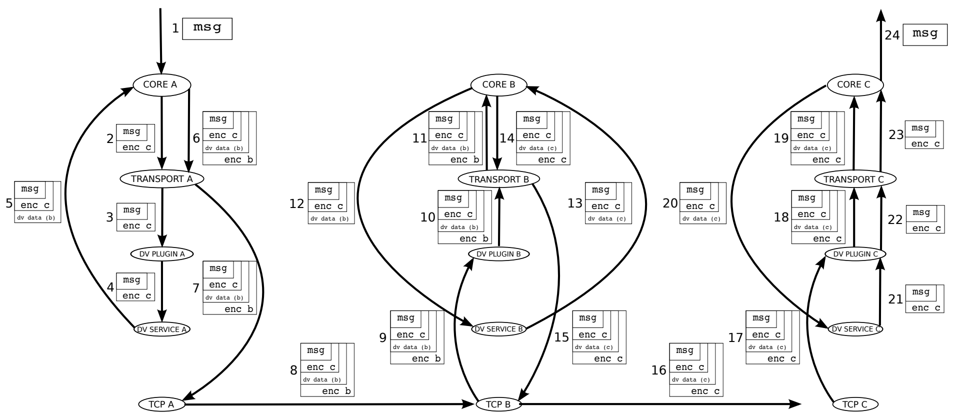


Fig. 5.2: Example showing details of the steps when sending a message from peer a to peer c via peer b.

In this particular example[1], transport (A) sends the doubly encrypted message to transport (B) via TCP. Peer B passes the message around, removing a single layer of encryption to reveal the distance vector data message specifying the message should be sent to C. Eventually a new distance vector data message is created and transport (B) sends the message to transport (C) via TCP. Once decrypted, the distance vector service (C) verifies that the original sender is valid and passes it to the distance vector plugin (C). The distance vector plugin mangles the message so that it appears to have arrived from peer A. The plugin gives this message to the transport service, which passes it to the core service of peer C, which finally decrypts the original message and passes it to whichever application(s) handle the message type.

5.5 Neighborhood Size Estimate

The purpose of our fish-eye bounded distance vector protocol is providing a greater number of peers to applications higher levels. The user-configurable component of this protocol is the fish-eye bound, or the number of hops distant the peer will consider to be its local neighborhood. This is an important parameter as the amount of storage overhead may grow exponentially when increasing the neighborhood bound. Specifically, if we assume a uniform topology where each peer has an average number α connections, Lemma 5.1 shows the maximum number of peers with varying fish-eye bounds.

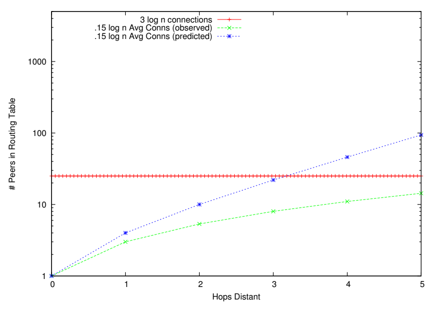
Remark 5.1. *Given a topology where peers have on average α direct connections and a fish-eye bound of β, the expected maximum fish-eye neighborhood*

[1] Any transport available connecting the two peers could be used

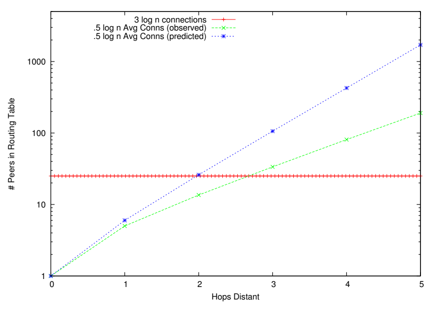
size, ignoring collisions, is:

$$\vartheta_{\alpha,\beta} = \sum_{h=0}^{\beta} \alpha^h \tag{5.1}$$

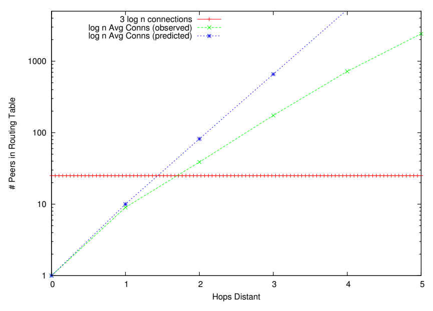
Given that the size of the fish-eye neighborhood can grow quite fast, we allow a user-configured limit on the size of the distance vector routing table. Once this specified limit is reached, new connections are only added if they are less costly than some already known reachable peer and a core level connection to that peer has not been established via the more costly link. If these conditions are met, the more expensive link entry is removed and the cheaper one added. Still, usage of the distance vector protocol should be carefully considered based on the expected topology that will be used. If used in a well connected topology, the distance vector transport will only use resources without providing any actual benefit.



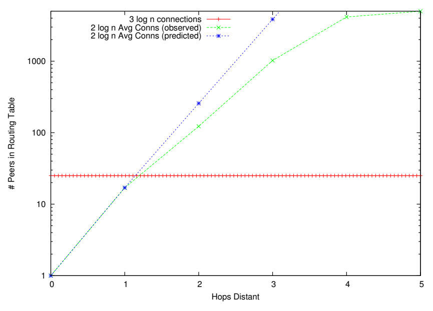
(a) $.15 \log(n)$ **average connections**



(b) $\frac{1}{2} \log(n)$ **average connections**



(c) $\log(n)$ **average connections**



(d) $2 \log(n)$ **average connections**

Fig. 5.3: Plot of the estimated size of the distance vector neighborhood when increasing the number of hops. Estimated vs. actual for a $5,000$ **peer Small-World topology with varying per-peer connections.**

Figure 5.3 relates the estimated number of peers in a distance vector routing table to the actual number of peers for varying neighborhood sizes in 5,000 peer Small-World topologies with varying average per-peer connections. The "observed" data comes from emulations running 5,000 peers. This shows the impact of collisions (finding the same peer in two different neighborhood steps) on the estimate that we have given in Lemma 5.1. Lemma 5.1 overestimates the peers reachable within a certain number of steps, as it ignores collisions. However, the figures show that even with collisions, we achieve significantly more than $3 \cdot \log n$ peers within a small number of hops. We use $3 \cdot \log n$ total connections as a target for FBDV as a simple point of reference between the different topologies. The main point is that within a small number of hops distant, FBDV provides This provides a sufficient number of peers for our DHT routing algorithm described in Chapter 7, which is ultimately the goal for the FBDV protocol.

5.6 Distance Vector for Onion Routing

The plugin and process isolation structure of GNUnet makes the distance vector implementation relatively complex, adding some steps that are not strictly necessary to perform neighbor-of-neighbor based routing. However, one nice side effect of this complexity is that distance vector messages are essentially onion routed between peers. Onion routing [71] is a method for achieving anonymous communication between two endpoints. This is achieved by multiply encrypting messages at the sender and forwarding the message through a number of peers in series to the receiver. Using traditional onion routing, the receiver can not easily discern the identity of the sender. In common anonymity providing systems [45, 117, 129, 190] the part of the sender and receiver identity that is hidden are the respective IP addresses. Typically, the fact that the sender and receiver are communicating is also hidden from adversarial third parties; those participating in the network or eavesdroppers. This property of anonymity is achieved through the multiple encryption techniques and the length of the anonymizing chain (also known as a "tunnel" or "circuit" depending on the system).

With onion routing, a message sender chooses a certain number of peers through which to relay the message. The sender then encrypts the message repeatedly, starting with the encryption key of the last peer in the chain and ending with the first peer. The message is then sent to the first peer which removes one layer of encryption and sends the message to the next peer in the chain until the message reaches the final peer, where the last layer of encryption is removed and the message is revealed. Encrypting the message in this way hides the content from each of the intermediate peers, and provided that each of the peers knows only the previous and next hop and the circuit is of suitable length (see [71] for details) it becomes difficult for adversaries to determine the sender and receiver with any degree of certainty. Figure 5.4 shows the layered encryption process in typical onion routing between a sender a and recipient d.

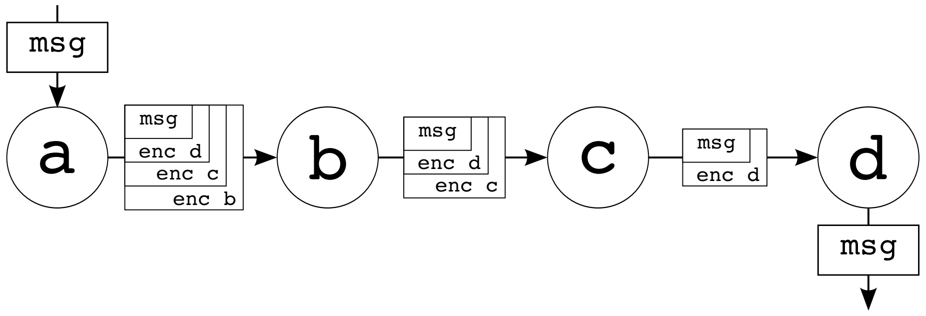


Fig. 5.4: Figure shows the high level view of a distance vector message being sent from peer a to peer d via peers b and c. Crucially for onion routing, the original message "msg" is encrypted thrice at peer a, and a single layer of encryption is peeled off at peers b, c and d, revealing the message only to the intended recipient.

Using the distance vector transport recursively, the exact same type of layered encryption is achieved, meaning that onion routing is performed implicitly. Figure 5.2 demonstrates how this works for a two hop distant peer using the distance vector transport, which is the most trivial example of onion routing. However, the only protection is from an eavesdropper, who would have to monitor both the $a \rightarrow b$ and $b \rightarrow c$ connections to discover that a was communicating with c. This becomes increasingly difficult for an eavesdropper as more peers are added between the sender and receiver. Figure 5.5 shows how a message would be constructed to be sent from a peer a to peer d provided that there are two hops b and c between them. The key to this technique is that the distance vector transport can be used recursively; meaning that a distance vector connected peer can provide other distance vector connected peers. This requires the message to go through the multiple phases of encryption which define onion routing.

5.7 FBDV Caveats: Onion Routing Without Anonymity

The primary purpose of the distance vector transport is to provide additional connectivity to peers, thus it is important to outline the differences between the de-facto onion routing we use and traditional onion routing or other mixing techniques. Tor [45] or I2P [190], for instance, use client selected peers for building anonymizing tunnels. Other designs, including Salsa [129], AP3 [117] and Cashmere [188] allow random tunnel participants to be chosen at each step. An important security goal of all of these designs is that peers involved in the tunnel are unable to discover the identity of both the initiator and recipient of the message. In our design there is currently no attempt to hide this information from participants in the tunnel. This means that a malicious participant in the local neighborhood which is used for routing

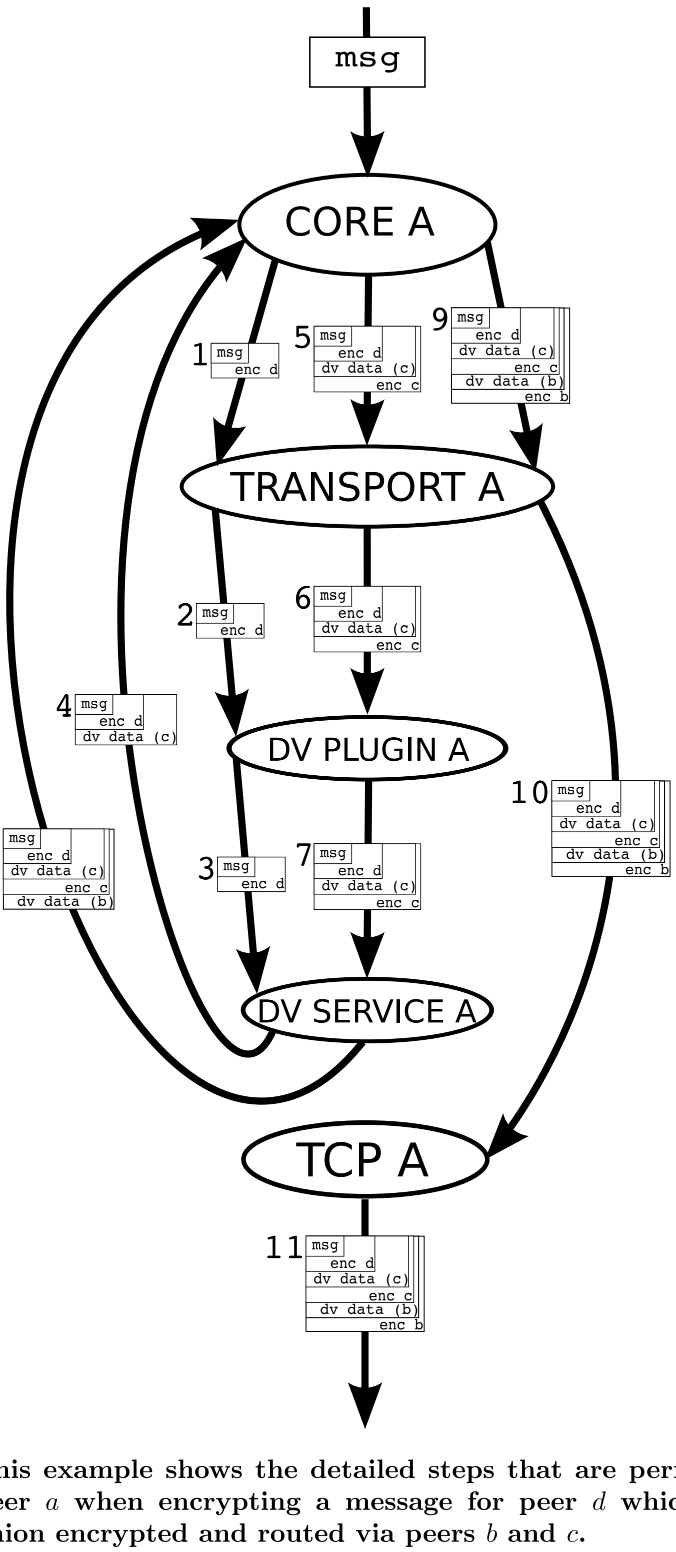
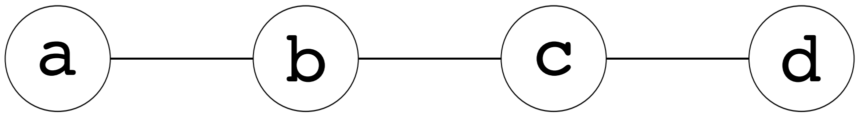

msg
CORE A
TRANSPORT A
DV PLUGIN A
DV SERVICE A
TCP A
1 msg enc d
2 msg enc d
3 msg enc d
4 msg enc d dv data (c)
5 msg enc d dv data (c) enc c
6 msg enc d dv data (c) enc c
7 msg enc d dv data (c) enc c
8 msg enc d dv data (c) enc c dv data (b)
9 msg enc d dv data (c) enc c dv data (b) enc b
10 msg enc d dv data (c) enc c dv data (b) enc b
11 msg enc d dv data (c) enc c dv data (b) enc b

Fig. 5.5: This example shows the detailed steps that are performed at peer a when encrypting a message for peer d which will be onion encrypted and routed via peers b and c.



	a	
peer	via	cost
c	b	2
d	b	3
d	c	3

	b	
peer	via	cost
d	c	2

	c	
peer	via	cost
a	b	2

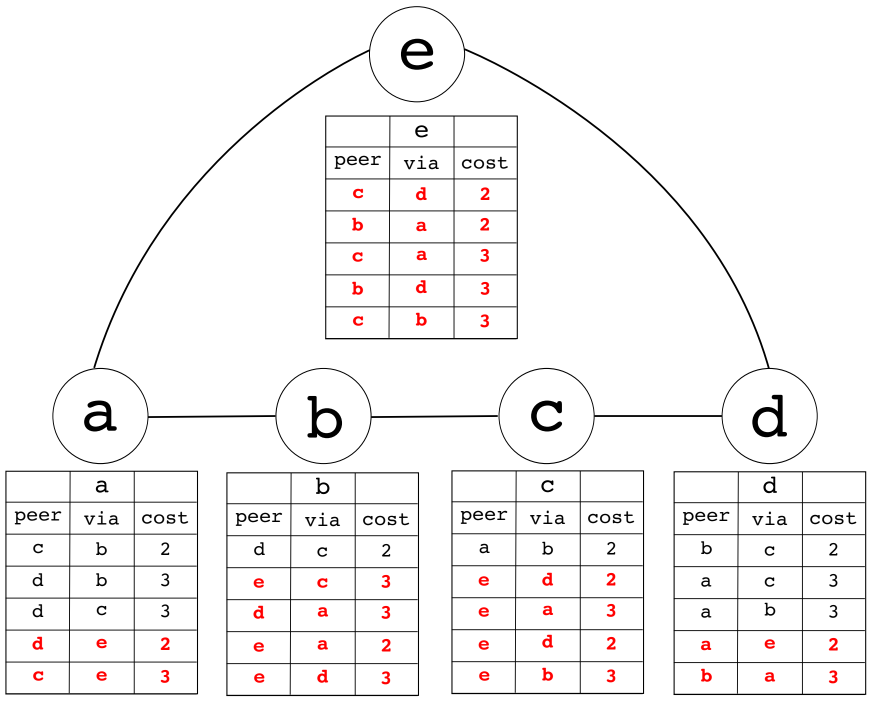
	d	
peer	via	cost
b	c	2
a	c	3
a	b	3

Fig. 5.6: Shown here are the routing tables for the distance vector service at the respective peers in this simple, straight line topology. The routing table for peer a contains two entries for peer d, one which will result in an onion encrypted connection and one which will not.

messages may know when two peers are communicating, though the contents of that communication remain secret. Thus the principal protection gained from onion routing with the distance vector transport is against outside eavesdroppers attempting to discover when two peers are communicating.

A second and equally important security aspect of other onion routing and mix network designs is that the peers chosen to participate in tunnels are chosen at random from the set of all peers in the network. This ensures that a small proportion of colluding malicious participants can not discover communication between peers trivially. Again, our design makes this requirement impossible. Essentially a peer is stuck with those neighbors within the fish-eye bounded range, be they well-behaved or malicious. A peer with even a single malicious neighbor providing connections to otherwise unaccessible peers will be likely to use the malicious peer. If the malicious peer is selected to be used, it could learn about communication patterns, drop messages intended for other peers, etc. This means that FBDV cannot even provide anonymity, and should not be assumed to do so.

Finally, while onion routing using the distance vector transport is currently possible, the default transport selection in GNUnet chooses transport plugins and addresses based on least cost and least latency. Take, for example, the topology in Figure 5.6. In it there are two entries in the routing table of peer a to contact peer d. Both traverse peers b and c, and both have an associated cost of 3. Both are equally likely to be chosen, but the first entry does not use recursive distance vector routing, and is therefore not onion encrypted. The second will use onion routing, but the transport selection mechanism in GNUnet has no way of knowing this (or preferring it). Even worse yet for security, if another peer e comes into the network providing a a connection to d for cost 2 (as shown in Figure 5.7) *that* connection will likely be chosen as the “best” link to use.[2] In its current state, the distance vector implementation cannot be used for anonymous routing, but it does meet our goal of providing additional connectivity among non-malicious peers.



	e	
peer	via	cost
c	**d**	**2**
b	**a**	**2**
c	**a**	**3**
b	**d**	**3**
c	**b**	**3**

	a	
peer	via	cost
c	b	2
d	b	3
d	c	3
d	**e**	**2**
c	**e**	**3**

	b	
peer	via	cost
d	c	2
e	**c**	**3**
d	**a**	**3**
e	**a**	**2**
e	**d**	**3**

	c	
peer	via	cost
a	b	2
e	**d**	**2**
e	**a**	**3**
e	**d**	**2**
e	**b**	**3**

	d	
peer	via	cost
b	c	2
a	c	3
a	b	3
a	**e**	**2**
b	**a**	**3**

Fig. 5.7: Here we show the (partial) routing tables for the topology shown in Figure 5.6; with a single additional peer which is directly connected to a and d. Distance vector connections added due to the topology change are shown in bold (red). The onion routed paths between peers are likely to be passed over in light of their higher cost.

5.8 Conclusion

We wanted a simple way to provide more connections to peers in restricted-route networks such as ad-hoc and wireless mesh networks. Combining traditional distance vector routing via a gossip protocol along with a fish-eye bounded local view suits our purposes of providing additional peers with low overhead. This enables peers utilizing the distance vector protocol to compute least cost links to nearby peers and provide them as connections to higher level applications, such as the DHT. This additional connectivity

[2] All of this depends on the transport selection algorithm being used; however the default is to use the shortest, fastest, highest bandwidth links. The result of this is that “short” distance vector paths will likely be faster as well as having shorter distances.

provides enough connections in even highly restricted Small-World and random networks for our DHT routing algorithm to perform well. As our design inherently provides onion routed connections in some circumstances, future work may allow high level applications to selectively choose which distance vector paths to use to achieve anonymity. Some of the security aspects of our design, specifically disallowing long circular routes, were put in place due to the large corpus of related work on onion routing and associated attacks (such as the one we show in Chapter 4).

The previous chapters have described our motivation for creating a new DHT routing algorithm, analyzed different extant P2P networks and introduced some of the methods employed for increasing connectivity in sparse networks. Another goal for our design is to create a usable implementation which can be deployed to a P2P network immediately, as opposed to a strictly theoretical design. We therefore also needed to provide a way to test our design, preferably without creating a separate implementation in a standalone simulator and attempting to merge the results back into the real-world implementation. Due to these concerns, we have also created an emulation framework for running many peers in a distributed manner. In addition to this general purpose emulation framework we also wrote specific software for testing the DHT, including a profiling driver, a web-based scheduler and result viewer for experimentation and some custom scripts for data processing. Therefore, before presenting our algorithm and describing evaluation results from experimentation, we describe our methodology for this evaluation and its implementation in the next chapter.

6. LARGE-SCALE DISTRIBUTED EMULATION OF P2P PROTOCOLS

The previous chapters have outlined the difficulty of creating a secure P2P routing algorithm for restricted-route networks. The next chapter covers our routing algorithm design which mitigates many of the problems in this domain while maintaining our goals. We have implemented this design in GNUnet, a P2P networking framework. While most new P2P routing algorithms are evaluated using simulation due to concerns of scalability and performance, simulation fails to capture implementation issues and commonly requires designers to create separate implementations for simulation and real-world deployment. Therefore, in order to test both our implementation as well as our design, we utilize emulation in our evaluation. To achieve suitable network sizes, we distribute emulation across multiple hosts. This chapter describes a distributed emulation framework for GNUnet that we have implemented so that we can perform large scale evaluations on our design in the following chapter. This chapter appeared in a different format at the 4th Workshop on Cyber Security Experimentation and Test [60].

6.1 Introduction

The outcome of a network security experiment can vary significantly depending on whether the experiment was based on simulation or emulation [29]. While both methods can provide new insights, there is a dearth of scalable approaches for assessing large-scale peer-to-peer (P2P) networks using emulation. While some studies [58] have run attacks against deployed P2P networks, there is a clear benefit to being able to run large-scale experiments without potentially negatively impacting actual users.

This chapter presents a design and implementation for large-scale experiments with P2P protocols using distributed emulation. The key insight is that while simulators can achieve significant scalability by abstraction, emulators for P2P networks can achieve comparable scalability through parallelization and distribution. By distributing computation, a modest computer laboratory can achieve performance gains of an order of magnitude over a single machine for suitable problems. Similarly, due to the advent of many-core processors, local computational resources are often only limited by the amount of parallelism in the problem. While parallelization and distribution of simulators can be difficult, distribution is inherent in P2P networking, making it easier to create distributed P2P emulators.

Emulation has many advantages over simulation: the code used for an experiment can be the same code used for deployment, and programming an appropriate model with abstractions is unnecessary. The emulator can be used to easily evaluate the entire system, not just small components; as a result, the experimental setup can be used to evaluate performance and security issues as well as serving as an integration testbed. Given problems on a deployed system, modifying experiments using emulation to reproduce, measure and evaluate the undesired behavior is generally easier than doing the same using experiments through simulation. In fact, depending on the abstractions chosen, the simulation may fail to model the problem observed in the real-world. This is even more applicable for security assessments as abstraction eliminates implementation details, and thereby sources of vulnerabilities.

P2P simulators typically simulate tens of thousands to hundreds of thousands of peers [76, 94, 120, 149], with some distributed simulators reportedly scaling up to 80 million peers [46]; however, while distributed emulation may be the natural choice for experiments with distributed systems, it is still not at all obvious that emulation actually would scale to the desired problem sizes. Previous work on emulation falls far short of the scale managed by simulations; the best-performing previous emulation setup that we are aware of has been reported to scale to only 4,096 peers [165].

This chapter details our design and experiences in providing a scalable framework for evaluating P2P protocols using emulation. We did experiments running a 80,000 peer Kademlia-style DHT (with link-encrypted P2P communication) using a small cluster of 32 machines with 7 GB of memory each; our results indicate that with proper tuning, emulation can be scaled to much larger sizes than previously achieved without sacrificing the ability to run realistic experiments.

6.2 Design Goals

The primary requirement for our experimentation framework is that it must be distributed, taking advantage of the inherent properties of P2P networks to spread computational load. Additionally, we required the ability to run many peers on the same host, making use of multiple cores and taking into account that a single peer would rarely use the entire computational or storage resources of an individual computer.

The main limitation of our design is that in order to achieve reasonable performance, we run peers as independent processes and assume no control over when the operating system gives CPU time to peers. As a result, our emulator cannot produce timing-accurate results. Emulation of network delays is not currently supported in our framework.

Given that an emulation is executed using a cluster (or single host) with universal connectivity, an important feature for realistic experiments is the ability to impose restrictions on which peers should be able to communicate directly. Existing P2P network simulators often only allow topology configuration at very low levels [95] such as configuring AS links or link delays. None of the DHT-capable simulators [1, 8, 120] that we have encountered allow user defined underlay topology restrictions, such as those which result from firewalls, network address translation (NAT) or topologies based on personal trust relationships, like those extracted from Facebook [182]. In contrast, our system can be configured to impose constraints on underlay topologies; we also provide various algorithms to construct common topologies such as cliques, Small-World networks and Internet-like graphs.

A primary goal of any experiment is to collect relevant data. Our system supports adding custom instrumentation to log results to file or to a database. However, given a sufficiently large number of data points, such logging activities can become the bottleneck for an experiment. To mitigate this potential issue, we provide a scalable integrated facility to log, accumulate and store simple numeric values collected during an experiment using a single function call. Collection of these statistics can be performed in a distributed or centralized manner.

Finally, one common criticism of most tools is that they have a steep learning curve, which can lead to strange results. The authors of [128] note that two different simulators running the same experiment on a five node Chord network produced inexplicably different results. Using emulation shifts this problem from understanding the simulator to understanding the underlying P2P system. The next section will describe key features of our P2P framework, arguing why we believe that this will ease usability problems.

6.3 Related Work

Large-scale testbeds such as DETER [116], GENI [61], SecSI [24], and Emulab [80] provide realistic network conditions and operating systems for security testing. Typically this research focuses on application level tests running full VMs [80] or OSes [29], revealing high-level performance and security [25] issues. In contrast, our design focuses specifically on P2P implementations, allowing security design issues to be discovered which may only be present at large scale. However, our emulation framework could easily be deployed on such testbeds as well to incorporate the effects of differences in the underlying platforms into the results.

6.3.1 Simulation

The prevailing method for testing and verifying new P2P designs is simulation [8, 46, 120, 161].

Chunksim [87] and the Query Cycle Simulator [161] are domain-specific discrete event simulators focusing on BitTorrent and content distribution, respectively. Both were created with the assumption that real-world experiments require the ability to model at least 20K peers and the presumption that this size of network could not be emulated or deployed for experiments. Furthermore, it was thought that malicious peers could not be adequately studied in deployed networks of this size. Our framework demonstrates that such limitations no longer hold.

OverSim [8] and PlanetSim [1] are discrete event simulators for overlay protocols. Both use a layered structure to provide a common API for application development. Overlay applications are written in a domain specific dialect of C++ (OverSim) or Java (PlanetSim); this high level of abstraction makes simulation implementations quite different from their counterpart real-world implementations. Building upon OMNet++ [180], OverSim is able to simulate down to the network level, achieving high realism. PlanetSim allows the networking layer to be switched out, allowing the use of a NetworkSimulator for simulation, and a NetworkWrapper for emulation or deployment. Simulations of up to 100K peers are reported, but we are not aware of any detailed studies that were performed at this scale.

Some effort has recently been made to distribute the tasks of existing simulators [46, 95, 166]. PeerSim and dPeerSim (the distributed version of PeerSim) are currently the most scalable P2P discrete event simulators with simulations of tens of millions of peers. While this is significantly larger than what we can do with emulation, the realism and freedom of implementation provided by our framework makes it complementary to these large-scale simulators.

6.3.2 Emulation

Emulation frameworks [81, 149, 165, 181] are less common than simulators. While emulators are typically better at testing real-world implementations and capture more realistic data, they do so at the expense of scalability. None of the existing emulation frameworks have been used for experiments at the scale presented in this chapter.

Contrary to our focus on high-level operations, ModelNet [181] is a distributed emulation environment aimed at capturing network delays, cross traffic and congestion due to underlay network configuration. In ModelNet, each emulated peer is executed with precise control over its network interactions. The largest ModelNet emulation included 10K Gnutella peers in a cluster of unspecified size, though full details of this particular experiment were not given in the research publication.

6.3.3 Combining Simulation and Emulation

Some projects try to overcome the scalability limitations of emulators and the undesirable effects of abstraction from simulators by mixing both techniques.

MACEDON [149] is a framework for the design, simulation and emulation of P2P algorithms. MACEDON requires that applications are written in a specific language; from this specification, code is generated for the simulator or emulator. MACEDON relies on ModelNet for underlay specification, and the largest reported experiment was less than 1K peers in total. We found no instances of MACEDON-generated code actually being used as a real-world implementation.

Other mixed simulation/emulation systems include Overlay Weaver [165] and RealPeer [81]. Overlay Weaver has been scaled up to 4K total peers on a 200 PC cluster. RealPeer is a framework and methodology for creating a P2P implementation in phases which include simulation and emulation. The Java-based implementation could conceivably be used as a real-world implementation. The authors successfully simulated a 20K node Gnutella network; however, no results were provided on testing the emulation or real-world implementation.

6.4 The GNUnet P2P Framework

A distinguishing characteristic of GNUnet is that each individual peer typically consists of roughly a dozen processes that use interprocess messaging. GNUnet uses one process for P2P communication, one process for the DHT, one for DNS resolution, and so on. Each peer's process group is coordinated using a master process which is primarily responsible for starting and stopping processes.

Minimization of memory consumption is critical for our system to scale. GNUnet is implemented in C, avoiding the large memory footprint of systems written using managed languages. C code is compiled prior to execution; hence, the operating system can use the same pages in real memory for the respective read-only code and data segments of different processes. Furthermore, memory consumption for the heap is reduced by avoiding garbage collection overheads.

This architecture has several key advantages. First, it isolates faults within components, making it easier to diagnose problems. Second, each peer can make use of many cores. Given that we run thousands of peers per host resulting in tens of thousands of processes, this architecture is suited to modern-day many-core processors [157]. Finally, a major advantage of the multi-process architecture is that new components can, in theory, be written

Tab. 6.1: List of GNU/Linux system settings that typically need to be reviewed for large-scale experiments.

/proc/sys/fs/file-max	System open files
/proc/sys/vm/swappiness	Swap preference
/etc/security/limits.conf	Per-user file limits
/proc/sys/kernel/pid_max	System processes
/etc/ssh/sshd.conf	SSH daemon limits
/proc/sys/net/core/rmem_max	UDP Buffer size

in other languages. While certain other languages may incur significant performance penalties for large-scale experiments, this facility may still be beneficial from a usability perspective.

When running thousands of peers on a single host (or millions of peers in a cluster), one key issue is that network and operating system limits for the number of processes, open sockets, and especially open TCP ports are easily exceeded. Table 6.1 lists some of the operating system options on GNU/Linux that typically need to be adjusted for large-scale experiments. It should be noted that GNUnet allows peers to communicate not only via TCP but also using UDP or even UNIX domain sockets, both of which can help skirt around operating system limitations. It is also possible to restrict connections between peers to only use particular transports, which is useful when emulating particular constraints on P2P communication.

Finally, we should mention that GNUnet provides some basic level of security for all applications using it. Each peer is identified via a public-private key pair (using 2048-bit RSA). Connections between peers are link-encrypted and authenticated using AES-256 and SHA-512. On the link-layer, network addresses are signed by participants, which ensures that peers do not send (encrypted) traffic to addresses other than those controlled by the intended destination. The network-layer also communicates and — as much as technically feasible — enforces bandwidth limitations as set by the user, and assigns bandwidth based on preferences as determined by the P2P applications using GNUnet.

6.5 The Emulation Library

We control large-scale emulation experiments using an experiment-specific driver that sets up the testbed using our emulation library. This library is accessed via a layered API. The low-level API provides functions to start and stop individual peers, to explicitly connect pairs of peers, and to change the configuration of running peers. For P2P security evaluations, the ability to dynamically reconfigure peers at runtime can be a valuable tool: in some of our experiments, we use it to configure a subset of the peers to become adversaries and start executing different attack vectors. The high-level API allows groups of peers to be started across multiple hosts, automatic generation of a range of network topologies and induction of network-wide churn at configurable rates. The API also provides means for accessing the state of each individual peer, including access to the exact current network topology and statistics logged by peers.

The high-level API is given a configuration file as a template for generating the initial configuration for each peer. The library itself primarily adjusts options such as port numbers that must be unique per peer. `ssh` is then used to copy configuration files to execution hosts and to start peers on those systems. The library assumes that login without password on hosts is possible via `ssh` (for example, using `ssh-agent`) and that the necessary binaries are installed in the PATH on the execution hosts.

6.5.1 Executing Experiments

Execution of an experiment using the library via the high-level API typically proceeds in six phases. During the first phase, a hostkey is generated for each peer; at this time, the testing driver is optionally notified of the identity (the hash of the public key) of each peer. This allows the controlling process to keep track of peer identities for later peer identification and lookup in the peer group. In the second phase, the desired network topology is computed and configuration files that specify the desired topological constraints are created. These constraints specify which connections are allowed at both the underlay and overlay level, but not which connections will actually be used. Peers are started in the third phase, and connected using the so-called *initial* network topology in the fourth phase. After all of these initial topology connections have been established, in the fifth phase, the driver performs actions specific to the experiment, during which time the network topology may evolve, configurations for individual peers can be modified, churn can be induced, and benchmarks may be executed. In the final phase, the emulation library is used to shut down all of the peers. Peers are shut down using (fresh) `ssh` connections to the execution hosts.

6.5.2 Peer Life Cycle

Whenever a peer startup request is received, a context for the new peer is created and the startup routine enters a finite state machine which handles the actual startup process. The minimum requirements for starting a single peer are a configuration handle (which has been initialized from a configuration file prior to being called) and a callback function to call once the peer has been started. Additionally the name of a host on which to run the peer, a callback function to call once the peer's public/private key pair has been

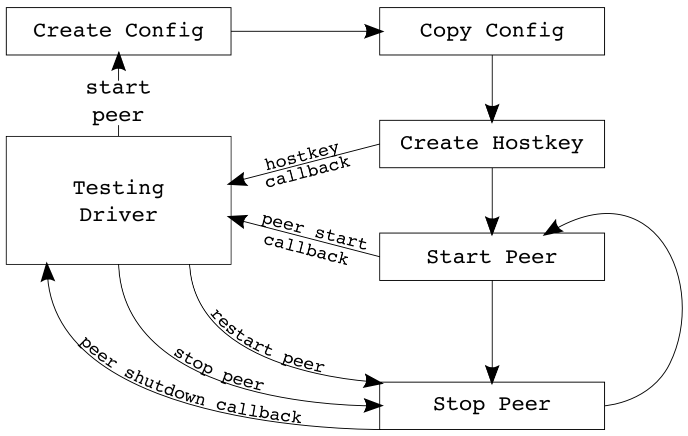


Fig. 6.1: Interactions that take place when a peer is started by a test driver.

generated, and other configuration options can be specified.

Figure 6.1 details the states and transitions that exist for a peer. First, a new configuration file (required for each started peer) is created and written to a temporary file on disk. This file is then copied to the proper (local or remote) directory. Next, the public/private key pair is generated and the testing driver is optionally notified. Then the peer is finally started, and once fully running, the testing driver is notified. Control remains with the testing driver until the peer is restarted or shut down.

6.5.3 Peer Group Life Cycle

A "peer group" in our testing framework provides a handle to a group of peers, and maintains per peer information internally on behalf of the testing driver process. The first step in a setting up a peer group is calling the `GNUNET_TESTING_daemons_start` function. This function requires a base configuration, the number of peers to be started, and callbacks for various notifications about the peer group startup process. Optionally a list of hosts (in hostname or IP address format) can be given such that the peers are evenly distributed over the hosts.

The peer group creation begins by creating one configuration file per peer locally, changing port numbers, paths and bind addresses as appropriate for each peer. Next, each configuration file is copied to the appropriate location for the host it will be run on. At this point, the framework enters a startup task which schedules the allowed number of peers to begin starting at each host. The normal peer life cycle applies here, as the single peer startup call is used to start each of the peers. However, peer group testing may be broken up into six distinct phases; hostkey generation, topology generation, actual peer startup, topology connection, test execution and peer shutdown.

The peer group startup is split into phases so that the testing driver can have as much (or as little) control over the peers as desired. The first phase is hostkey generation, at which point the testing driver is optionally notified of the peer identity of each peer. This allows the controlling process to keep track of peer identities for later peer identification and lookup in the peer group. Once the hostkeys have been generated the testing framework has enough information to create an underlay topology on the network. As described further in Section 6.5.4, GNUnet uses friend files and transport level blacklists to restrict underlay topologies. If these are used, they need to be set up prior to peer startup to ensure the restrictions are properly enforced. At this point, the testing driver may specify the desired topology options, the topology is internally generated and the necessary files are copied to the hosts which will run the peers. After topology generation, the framework enters the peer startup phase, where peer startup is performed incrementally as limited by local and remote host constraints. As each peer is started, the testing driver is optionally informed of each configuration; otherwise a single notification is called once all peers have been started. Depending on the peer configuration, peers may be started without any connections to other peers. In this case, the testing driver then enters the topology connection phase, where explicit connections between peers specified by the topology are created. A user specified maximum number of outstanding connections are scheduled per host, and as each connection is made the testing driver is (again, optionally) notified. The driver may or may not need to know the details of the created topology, depending on what the test is used for. After all topology connections have been established, control is returned to the testing driver for the test execution phase. When the testing driver has finished execution, the testing framework enters the final peer shutdown phase. Peers are incrementally shutdown in the same fashion as they were started, throttling the process in accordance with user specified limits.

6.5.4 Topology

One of the major goals for our testing framework was to be able to connect peers in a diverse set of specific topologies. We also wanted to distinguish easily between the underlay topology, which transport level connections are allowed, and the overlay topology, which peer level connections are allowed. White-listed peers (those explicitly allowed to connect) and blacklisted peers (those explicitly disallowed) are also both supported. Finally, it is not uncommon for applications to include the bootstrapping of the network (i.e. connecting peers) as part of the protocol. We accomplish all these goals by allowing the user to specify up to three different topologies in the topology creation phase.

The first topology that can be specified when starting peers in a peer group is called the "allowed topology". This topology is used as the base topology for connecting peers if no other options or topologies are used. If overlay white-listing is enabled, friend files are created per peer listing those that are allowed to connect. This will restrict overlay connections to only those peers, regardless of what are later attempted.

The second topology that can be specified is the "blacklist topology". This topology is used to disallow underlay connections for *specific* transport plugins. The transports to blacklist are given as a space delimited list when creating the topology. Peer and transport plugin combinations that match against the blacklist are disallowed, even if the same peer is white-listed at the overlay level.

The final topology that can be specified is the "initial topology". This topology is used mainly for two reasons; the first is that while a certain topology is allowed, not all the connections should be made initially. This covers, for instance, testing peer discovery mechanisms, without otherwise restricting the topology. The second reason is that using the initial topology when there is no other mechanism for peer discovery is an easy way to restrict peer connections (without resorting to blacklisting).

Our framework supports specification of each of the topologies listed in Table 6.2. Our intent is to provide well-known topology generators for experiments involving both highly structured and random topologies. Note that some of the supported topology generation algorithms require specification of arguments to control their construction. For example, randomized graph construction requires an argument indicating the probability for establishing a link. All of the options and their arguments are listed in Table 6.3.

The "file" topology is the most expressive option supported. This topology requires an argument indicating the path to a topology specified in the METIS [89] file format. We allow topologies to be specified in these files both for easy import from other topology generators and for consistency in testing. The emulation library provides utility functions to export the topology of an active peer group to file. This file can be used to recreate the exact same topology during subsequent experiments, or repetitions of the same experiment.

6.6 Lessons Learned

The development process for our system was iterative: as we scaled up the network size, new bottlenecks would emerge and had to be dealt with before

Topology	**Description**	Percentage	Probability
Clique	Connects all peers	No	No
Line	Connect peers in line	No	No
Ring	Line with wraparound	No	No
2d-Torus	2d-grid with wraparound	No	No
Erdős-Rényi	Random graph construction	No	Yes
Small-World (ring)	Ring topology with extra long distance connections	Yes	Yes
Small-World (2d-torus)	2d-torus topology with extra long distance connections	Yes	Yes
Scale Free	Scale free topology	No	No
InterNAT	Clique with percentage disallowed	Yes	No
File	Read topology from file	No	No

Tab. 6.2: Generators for the allowed, blacklist and initial topologies.

Option	**Description**	**Modifier**
Closest	Connect closest (XOR distance) peers	Number to connect
Random Subset	Random portion of allowed connections	–
DFS	Depth first traversal from random starting point	Minimum connections per peer
All	All possible connections	–
Minimum	Every peer has at least n connections	Number to connect

Tab. 6.3: Additional options for constraining initial peers in the initial topology.

we were able to increase the size again. This section summarizes the most salient lessons learned about emulating large-scale P2P networks.

6.6.1 Cryptography

Cryptographic operations, in general, can be expensive, and given that many modern P2P networks use asymmetric key pairs for host identification and other pieces of core functionality, repeated calls to cryptographic functions (e.g. key generation) can have significant overhead.

Our experience during development was that the creation of strong private keys, even for just a few dozen peers, virtually always depleted the entropy pool of the system, causing excessive delays. Our initial response was to disable entropy gathering; however, creating tens of thousands of 2048-bit RSA keys for each experiment still took a significant amount of time, even using a cluster.

We solved this problem by pre-computing the public-private key pairs for all peers and reusing them between experiments. It should be noted that despite the use of rather expensive cryptographic primitives, we did not have to simplify or eliminate other cryptographic operations.

For emulating P2P systems that require cryptography, there are two lessons to be learned here. First, strong key-generation operations (which generally have little impact for end-users in terms of system performance) need to be simplified even for small-scale emulation experiments. Second, assuming the protocols are reasonably well-designed, other typical cryptographic operations do not need to be simplified even for large-scale emulation experiments.

6.6.2 Execution time

One important discovery we made was that when running tests at a large scale, tasks that need to run at a fixed frequency are problematic, especially if their number increases with an experimental variable.

When crossing the 5,000 peer threshold, our experiments were stuck at a total of 200K connections or less; any increase caused the connection process (and our test host) to grind to a halt. The reason, we discovered, was that the CPU became pegged processing latency estimation tasks. These tasks were initially set to run at a seemingly harmless frequency of once per minute per connection. With around 40 connections per peer and 5,000 peers, these tasks became backed up, effectively taking over the system. In this case, our simple solution was to allow users to decrease the frequency of the latency measurements when testing, which is harmless as our emulation setup does not model network latencies (and, thus, these measurements would not be accurate anyway).

Because the overall issue is that in a general-purpose framework, virtually any hard coded value will eventually cause problems, the solution is to either make the code adaptive — for example, our system uses exponential back-offs in many places instead of fixed retry frequencies — or at least configurable by users (preferably with clearly marked default configuration values).

6.6.3 Latency

One important general lesson we learned in this endeavor is that while parallelism is important for avoiding idle waiting and efficient utilization of available resources, it must always be bounded and balanced in order to avoid overly negative impacts on latency. More specifically, we discovered that once we moved to distributed operations, the latency of starting peers over the network quickly became a major bottleneck, as creating an `ssh` connection and waiting for complete startup of a single peer took up to a second — even without key generation.

To solve this issue, we first reduced the startup delay by launching only the master process (Section 6.4), and not waiting until the peer fully initialized. Interactions with individual peers were deferred until after all peers had been launched. Still, having to interact with each execution host for each peer created unacceptable delays of many network round-trip times. Ultimately, the latency issue for starting peers was resolved by creating a helper script that would start all the necessary peers on a particular host; using this script, we only require one `ssh` interaction per host instead of one per peer.

Additionally, because the creation of an initial set of connections between peers requires the driver to communicate the desired initial connections to those peers, another general latency issue we discovered was attempting to establish too many simultaneous connections at once. In the case of initial connection setup by the driver, we cannot fully parallelize this step, as the driver would run out of file handles. Furthermore, in general, trying to establish too many connections at a single host in parallel can peg the CPU on that host while other hosts remain idle. To solve this, our driver currently imposes configurable limits on the number of concurrent connection attempts per peer and per host; this ensures that all hosts are utilized until all connections are established.

By bounding and balancing our use of parallelism and by reducing the overall number of round-trip times during network operations (in our case, done by introducing the helper script), it is possible to effectively control many of the latency issues introduced by distribution.

6.6.4 Sockets

Attempts to maintain network transparency also created issues. As we scaled our implementation, we repeatedly ran into socket limitations. First, we ran out of TCP ports because each peer used a TCP port for every service. After switching to UNIX domain sockets for each peer's internal interprocess communication, we were still exceeding operating system limits when creating hundreds of thousands of TCP connections between peers on the same host. We had initially expected that UDP would be a good alternative, but quickly discovered that UDP becomes highly unreliable even over loopback once the kernel's UDP buffer becomes too small to handle all queued messages.

As a result, our large-scale experiments are typically configured to use UNIX domain sockets for all inter-peer communication on the same host and UDP or TCP between peers. Furthermore, TCP-based control connections between the driver and peers are established on-demand.

The lesson we learned here is that while network transparency is nice (and in fact sometimes required for interactions with the driver), using UNIX domain sockets instead of TCP/UDP wherever possible is important for scalability. Being able to choose the communication domain is important, and the most scalable end-result is typically a mixture of UNIX, UDP and TCP.

6.6.5 Memory

When running tens of thousands of peers, every single additional private page is costly; as such, a repeatedly recurring bottleneck was memory consumption. While many specific changes to data structures were made to reduce memory consumption, the single biggest improvement was obtained by changing the size of our communication buffers from a static 64k bytes to a minimum initial size of 4 bytes, which is then re-allocated to a larger number as necessary. Most messages which are passed between services or peers are much smaller than 64k, so this change resulted in significant memory reductions for most services.

Per design, each of our peers is comprised of a number of services implemented as independent processes. Depending on the nature of the experiment, some of these service processes can be shared between peers. For example, sharing the DNS resolution service is typically unproblematic as it has no state. We typically share one instance of certain services among every 100 peers. This reduces the overall memory footprint without turning these shared processes into bottlenecks.

The general conclusion here is that being able to share memory between peers is critical for large-scale emulation, and that any non-shared memory (including heap, stack and pages for global variables) must be closely inspected. We have been able to push the amount of memory shared between peers to about 80%, which represents an improvement in scalability by a factor of five.

6.7 Results

We have extensively tested our emulation framework on various architectures under myriad configurations. In this section we present some basic performance data for the framework and a small selection of the results obtained from our experiments.

Table 6.4 shows the results of our framework in terms of scalability (measured in the number of peers emulated) and topology setup times for various architectures and system configurations. In terms of time, the most costly part of the emulation set up is typically the peer connection phase, as it requires cryptographic key exchange and often network communication. The speed at which peers can be connected to each other is therefore an important metric.

Architecture	# Hosts (Total)	Cores (Total)	Memory (Total)	Max Peers	Conns per/s	Time to start peer
ARMv7 Cortex-A8	1	1	512 MB	100	∼ 1	∼ 206 ms
Xeon W3505	1	2	12 GB	2,025	∼ 60	∼ 12 ms
Xeon W3520	1	8	12 GB	2,025	∼ 188	∼ 5 ms
AMD Opteron 8222	1	16	64 GB	10,000	∼ 327	∼ 27 ms
AMD Opteron 850	31	124	217 GB	80,000	∼ 559	∼ 1 ms

Tab. 6.4: Relevant performance details of our framework on various architectures.

Service	Non-shared	Heap	Shared
supervisor	228 KB	32 KB	2,364 KB
transport	359 KB	99 KB	2,888 KB
core	300 KB	84 KB	2,428 KB
dht	536 KB	240 KB	3,684 KB
total	1,424 KB	456 KB	11,364 KB

Tab. 6.5: Breakdown of memory usage for the relevant services extant in our framework. Each process uses an additional 84 KB for the stack. P2P connections require about 6 KB memory with the transport service, 7 KB with the core service and an additional 1 KB if the DHT is using the connection. Our code was compiled on a 64-bit Linux system using GNU GCC version 4.3.2 optimized for size ("gcc -Os").

The data shows that our framework is quite scalable, running up to 100 peers on an (embedded) ARMv7 device and 80,000 on a small cluster. As one would expect, the number of peers we can emulate correlates closely with the total amount of system memory and the connection speed relates to the processor speed. We also see that peer startup time is reduced as more cores are added, with the notable exception of the 16-core host, with a comparably long peer startup process. We believe this is due to IO limitations; starting each peer requires peer-specific file accesses to the configuration and private key file.

The most important factor in terms of scalability of our framework is the memory footprint of the processes that make up each individual peer. Table 6.5 shows how memory is used by the various processes that each of our peers typically uses.

6.8 DHT Profiler Details

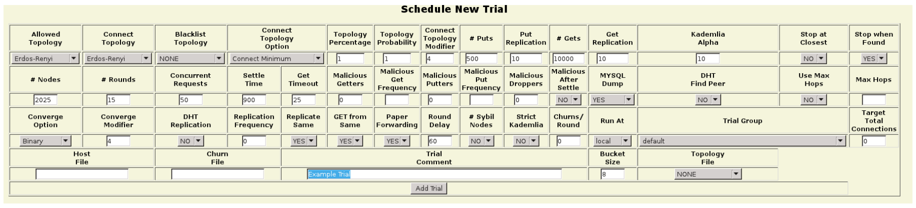
The DHT profiler utilizes the GNUnet testing framework for running performance and profiling tests on R^5N, the GNUnet specific DHT implementation detailed in Chapter 7. It is made up of a number of individual components, each listed below. The main goal of the profiler is to allow users to test a DHT implementation by easily setting up a series of experiments that cover a range of scenarios. These trials are automatically executed, the results processed and finally displayed to the user. We have tried to incorporate many standard metrics while leaving the original data intact for other processing to be performed. The main components that make up the DHT profiler are:

- Web Trial Scheduling - Used for choosing parameters for trials to be run, front-end to database
- Trial Execution Daemon - Reads from the database of trials to run, creates temporary configurations and runs tests
- Profiling Driver - GNUnet binary that creates a topology, then issues rounds of DHT *PUT* requests and *GET* requests
- Trial Processing Scripts - Executed once daemon finishes trial, inserts result to database
- Web Result Processing/Viewing/Comparison - Front end for viewing results
- Database Back-end - Stores all data describing trials and their results
- Database Interaction (Data output) - Instrumentation within GNUnet to output data to database

6.8.1 Web Trial Scheduling

In order to make scheduling trials and evaluation, as well as debugging, as easy as possible we have created a dynamic web interface. This interface has two main components, the web based trial scheduler (and background daemon), and the trial processing/information display. These two tools make it much simpler to perform evaluations on different possible set ups.

Figure 6.2 shows the web page used for viewing and scheduling upcoming trials. This interface is minimalistic, the page is populated directly from a database table updated whenever information for a future desired trial is entered. The trial daemon uses this database table to run trials using the GNUnet profiling driver.



Schedule New Trial

Allowed Topology	Connect Topology	Blacklist Topology	Connect Topology Option		Topology Percentage	Topology Probability	Connect Topology Modifier	# Puts	Put Replication	# Gets	Get Replication	Kademlia Alpha	Stop at Closest	Stop when Found
Erdos-Renyi	Erdos-Renyi	NONE	Connect Minimum		1	1	4	500	10	10000	10	10	NO	YES
# Nodes	# Rounds	Concurrent Requests	Settle Time	Get Timeout	Malicious Getters	Malicious Get Frequency	Malicious Putters	Malicious Put Frequency	Malicious Droppers	Malicious After Settle	MYSQL Dump	DHT Find Peer	Use Max Hops	Max Hops
2025	15	50	900	25	0		0		0	NO	YES	NO	NO	
Converge Option	Converge Modifier	DHT Replication	Replication Frequency	Replicate Same	GET from Same	Paper Forwarding	Round Delay	# Sybil Nodes	Strict Kademlia	Churns/ Round	Run At	Trial Group		Target Total Connections
Binary	4	NO	0	YES	YES	YES	60	NO	NO	0	local	default		0
Host File		Churn File	Trial Comment								Bucket Size	Topology File		
			Example Trial								8	NONE		

Add Trial

Fig. 6.2: The web page used for scheduling trials to be run.

6.8.2 Trial Execution Daemon

The trial execution daemon is a tool run on hosts that are executing trials via the DHT profiler. This daemon is a Perl program that reads from the scheduled trial database table. From this the daemon creates a meta-configuration file, and executes the DHT profiler with this file. After the trial is completed, the daemon collects results, updates the database, deletes intermediate files and directories and continues with the next trial.

For smaller trials the profiling driver and each of the GNUnet peers can directly communicate with the database that stores all of the data that is produced by running the trial (see Section 6.8.7). When this is not feasible, or has an effect on the performance of peers, data is dumped to files. In this configuration, the trial execution daemon must export this data to the database server. This involves reordering data as well as reformatting it to reduce database communication overhead.

6.8.3 Profiling Driver

The DHT profiling driver provides the core functionality for DHT testing. It is executed with the path to a configuration file which acts as the base file for all started peers as well as containing the meta data to actually run the tests as specified in the trial scheduling interface.

First, the configuration options specified in Table 6.6 are read from the configuration into memory. The driver then starts the required number of peers using the testing framework. Once all peers are running and a topology has been created, the driver then either waits the requisite "settle time" to allow the network to further bootstrap itself, or begins initiating *FIND PEER* requests for the DHT peers.

Once the settle time has expired, the bootstrapping phase of testing the DHT is finished, and the actual data gathering begins. We have chosen to split up this testing into a series of rounds; in each round some combination of *PUT* and *GET* messages are issued and the results are tallied.

6.8.4 Additional Trial Processing

Depending on the logging level specified when the trial was run, additional processing on the output data is required. This secondary data can be generated on the fly when using the web interface to view results, but generating this data can cause unseemly delays when navigating the interface. Therefore, we created a script which does most of the CPU intensive data crunching and writes the results to secondary database tables.

The base logging level records only the total number of *PUT* and *GET* requests that are issued and the resulting number of successful *GET* requests in each round. This is inexpensive; adding no overhead to peers. At the second logging level, the initiating and terminating peer(s) are logged for every

Option	Description
Allowed Topology	Peers may connect in this topology
Initial Topology	Explicitly create connections in this topology
Blacklist Topology	Disallow connections in this topology
Topology Percentage	Modifier for certain topologies
Topology Probability	Modifier for random topologies
Connect Modifier	Modifier for initial topology
# Puts	Number of *PUT* operations (per round)
# Gets	Number of *GET* operations (per round)
Put Replication	r value to set for *PUT* operations
Get Replication	r value to set for *GET* operations
Kademlia α	Parallel PUTs/GETs (Kademlia routing only)
Stop at Closest	Cease routing once nearest peer found
Stop when Found	Cease routing once data found (GET only)
# Nodes	Number of peers to emulate
# Rounds	Number of rounds of PUTs/GETs to perform
Concurrency	Number of PUTs/GETs allowed at a time
Settle Time	Seconds to allow for *FIND PEER* messages
GET Timeout	Seconds to wait for a *GET* response
Malicious Getters	Number of peers flooding *GET* requests
Malicious *GET* frequency	How often to flood *GET* requests
Malicious Putters	Number of peers flooding *PUT* requests
Malicious *GET* frequency	How often to flood *PUT* requests
Malicious Droppers	Number of peers dropping all requests
Malicious After Settle	Allow topology stabilization for malicious peers
SQL Dump	Dump detailed, minimal or no data to SQL server
DHT Find Peer	Allow each peer to automatically send *FIND PEER* messages
Max Hops	End requests after set number of hops
Converge Option	Choice of routing algorithm randomization
Converge Modifier	c parameter for routing algorithm
DHT Replication	Peers manage replication internally
Replication Frequency	How often peers replicate stored items
Replicate Same	Start *PUT* requests at same peer in each round
Get from Same	Start *GET* requests at same peer in each round
Round Delay	Delay some number of seconds between each round
Sybil Nodes	Choose closest peers for malicious
Strict Kademlia	Use recursive Kademlia routing
Churn per Rounds	Number of peers to churn on/off per round
Run At	Which testing driver peer to run on
Trial Group	Name of group to assign this test to
Target Connections	Number of connections to allow in the network
Host File	File name to read list of hosts from
Trial Comment	Comment for this trial
Bucket Size	Size of routing table buckets
Topology File	File to read topology information from

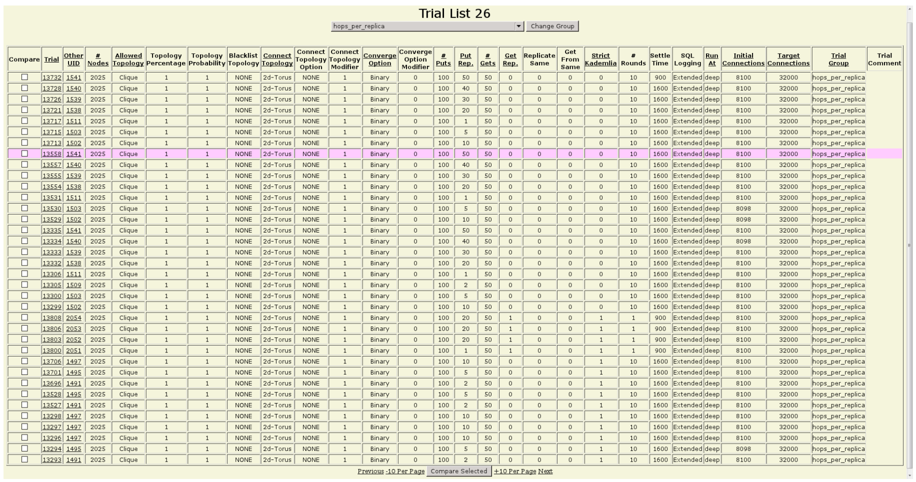
Tab. 6.6: Options that are available for configuring a trial to run.

request. This includes *FIND PEER* requests and *GET* reply messages in addition to *PUT* and *GET* requests. This added information provides success rates for *PUT* requests and *PUT* and *GET* replication. The number of hops that were traversed to reach the destination peer (but not the intermediate path) can also be determined. At this level the peer identities present in the network and the routing tables of each peer are logged at selected intervals. Thus, the network can be recreated from virtually any point in time throughout the duration of the trial. Also, plots of topology generation can be made, and other calculations such as the number of nearest peers in the network to a particular key, average peer connectivity, etc.

The highest level of logging records each hop of each request as it traverses the network. Also logged are which peers are *intended* to be routed to at each step; enabling detection of packet or core level message failures. Recording the full routes of *PUT*, *GET* and *GET* reply requests enables visualization of these routes. Furthermore, this information provides the total number of hops that are traversed in the course of a request, instead of just the hops to the peer(s) where the request terminated.

6.8.5 Web Result Processing/Viewing/Comparison

Figure 6.3 shows the web page for navigating through previously executed trials. From this overview page, trial meta data can be viewed and the trial can be selected for further inspection. Trial summary information is displayed; the trial ID field is a link which leads to a full trial detail page.

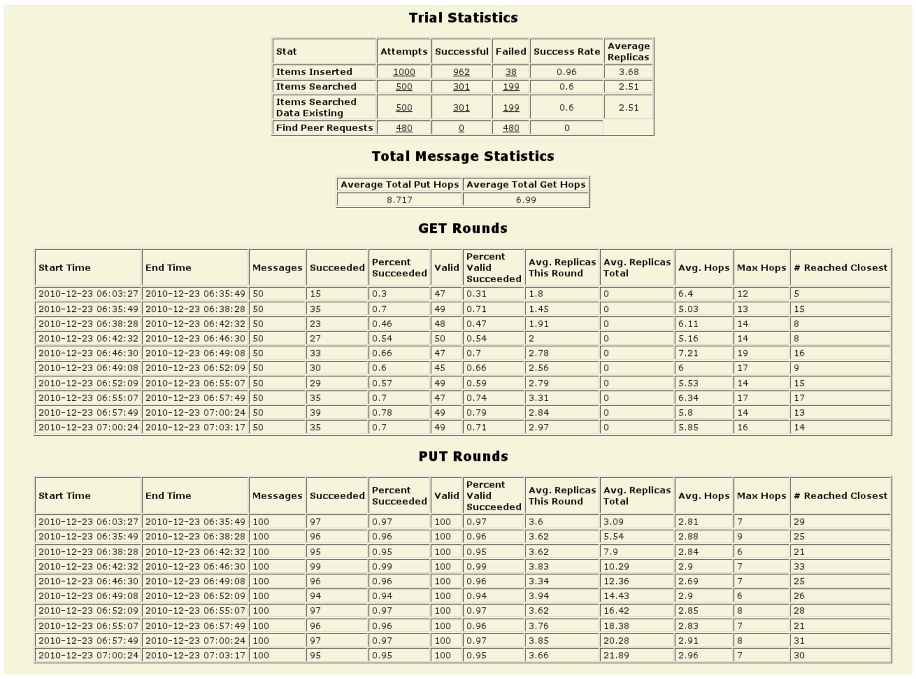


Compare	Trial	Other UID	# Nodes	Allowed Topology	Topology Percentage	Topology Probability	Blacklist Topology	Connect Topology	Connect Topology Option	Connect Topology Modifier	Converge Option	Converge Option Modifier	# Puts	Put Rep.	# Gets	Get Rep.	Replicate Same	Get From Same	Strict Kademlia	# Rounds	Settle Time	SQL Logging	Run At	Initial Connections	Target Connections	Trial Group	Trial Comment
☐	13732	1541	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	50	50	0	0	0	0	10	900	Extended	deep	8100	32000	hops_per_replica	
☐	13728	1540	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	40	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13726	1539	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	30	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13721	1538	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	20	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13717	1511	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	1	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13715	1503	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	5	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13713	1502	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	10	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13558	1541	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	50	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13557	1540	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	40	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13555	1539	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	30	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13554	1538	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	20	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13531	1511	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	1	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13530	1503	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	5	50	0	0	0	0	10	1600	Extended	deep	8098	32000	hops_per_replica	
☐	13529	1502	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	10	50	0	0	0	0	10	1600	Extended	deep	8098	32000	hops_per_replica	
☐	13335	1541	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	50	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13334	1540	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	40	50	0	0	0	0	10	1600	Extended	deep	8098	32000	hops_per_replica	
☐	13333	1539	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	30	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13332	1538	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	20	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13306	1511	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	1	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13305	1509	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	2	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13300	1503	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	5	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13299	1502	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	10	50	0	0	0	0	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13808	2054	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	20	50	1	0	0	1	1	900	Extended	deep	8100	32000	hops_per_replica	
☐	13806	2053	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	20	50	1	0	0	1	1	900	Extended	deep	8100	32000	hops_per_replica	
☐	13803	2052	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	20	50	1	0	0	1	1	900	Extended	deep	8100	32000	hops_per_replica	
☐	13800	2051	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	1	50	1	0	0	1	1	900	Extended	deep	8100	32000	hops_per_replica	
☐	13706	1497	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	10	50	0	0	0	1	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13701	1495	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	5	50	0	0	0	1	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13696	1491	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	2	50	0	0	0	1	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13528	1495	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	5	50	0	0	0	1	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13527	1491	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	2	50	0	0	0	1	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13298	1497	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	10	50	0	0	0	1	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13297	1497	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	10	50	0	0	0	1	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13296	1497	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	10	50	0	0	0	1	10	1600	Extended	deep	8100	32000	hops_per_replica	
☐	13294	1495	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	5	50	0	0	0	1	10	1600	Extended	deep	8098	32000	hops_per_replica	
☐	13293	1491	2025	Clique	1	1	NONE	2d-Torus	NONE	1	Binary	0	100	2	50	0	0	0	1	10	1600	Extended	deep	8100	32000	hops_per_replica	

Fig. 6.3: The web page used for displaying already run trial information.

Figure 6.4 shows this trial detail page. This allows users to quickly view results of trial the including performance, efficiency, and so forth. A trial statistics table is shown that lists the number of requests issued and the success rates for each. A table of malicious message statistics is provided showing how these messages affect performance. Efficiency statistics including the hops required for different message types is also provided. Each specific trial metric links to additional web pages displaying details about the metric. From these more specific pages, users can drill down to individual routing information for each message (depending on the logging level).

For instance, there were 38 failed *PUT* (Items Inserted) messages in the trial displayed in Figure 6.4. Clicking "38" presents the page shown in



Trial Statistics

Stat	Attempts	Successful	Failed	Success Rate	Average Replicas
Items Inserted	1000	962	38	0.96	3.68
Items Searched	500	301	199	0.6	2.51
Items Searched Data Existing	500	301	199	0.6	2.51
Find Peer Requests	480	0	480	0	

Total Message Statistics

Average Total Put Hops	Average Total Get Hops
8.717	6.99

GET Rounds

Start Time	End Time	Messages	Succeeded	Percent Succeeded	Valid	Percent Valid Succeeded	Avg. Replicas This Round	Avg. Replicas Total	Avg. Hops	Max Hops	# Reached Closest
2010-12-23 06:03:27	2010-12-23 06:35:49	50	15	0.3	47	0.31	1.8	0	6.4	12	5
2010-12-23 06:35:49	2010-12-23 06:38:28	50	35	0.7	49	0.71	1.45	0	5.03	13	15
2010-12-23 06:38:28	2010-12-23 06:42:32	50	23	0.46	48	0.47	1.91	0	6.11	14	8
2010-12-23 06:42:32	2010-12-23 06:46:30	50	27	0.54	50	0.54	2	0	5.16	14	8
2010-12-23 06:46:30	2010-12-23 06:49:08	50	33	0.66	47	0.7	2.78	0	7.21	19	16
2010-12-23 06:49:08	2010-12-23 06:52:09	50	30	0.6	45	0.66	2.56	0	6	17	9
2010-12-23 06:52:09	2010-12-23 06:55:07	50	29	0.57	49	0.59	2.79	0	5.53	14	15
2010-12-23 06:55:07	2010-12-23 06:57:49	50	35	0.7	47	0.74	3.31	0	6.34	17	17
2010-12-23 06:57:49	2010-12-23 07:00:24	50	39	0.78	49	0.79	2.84	0	5.8	14	13
2010-12-23 07:00:24	2010-12-23 07:03:17	50	35	0.7	49	0.71	2.97	0	5.85	16	14

PUT Rounds

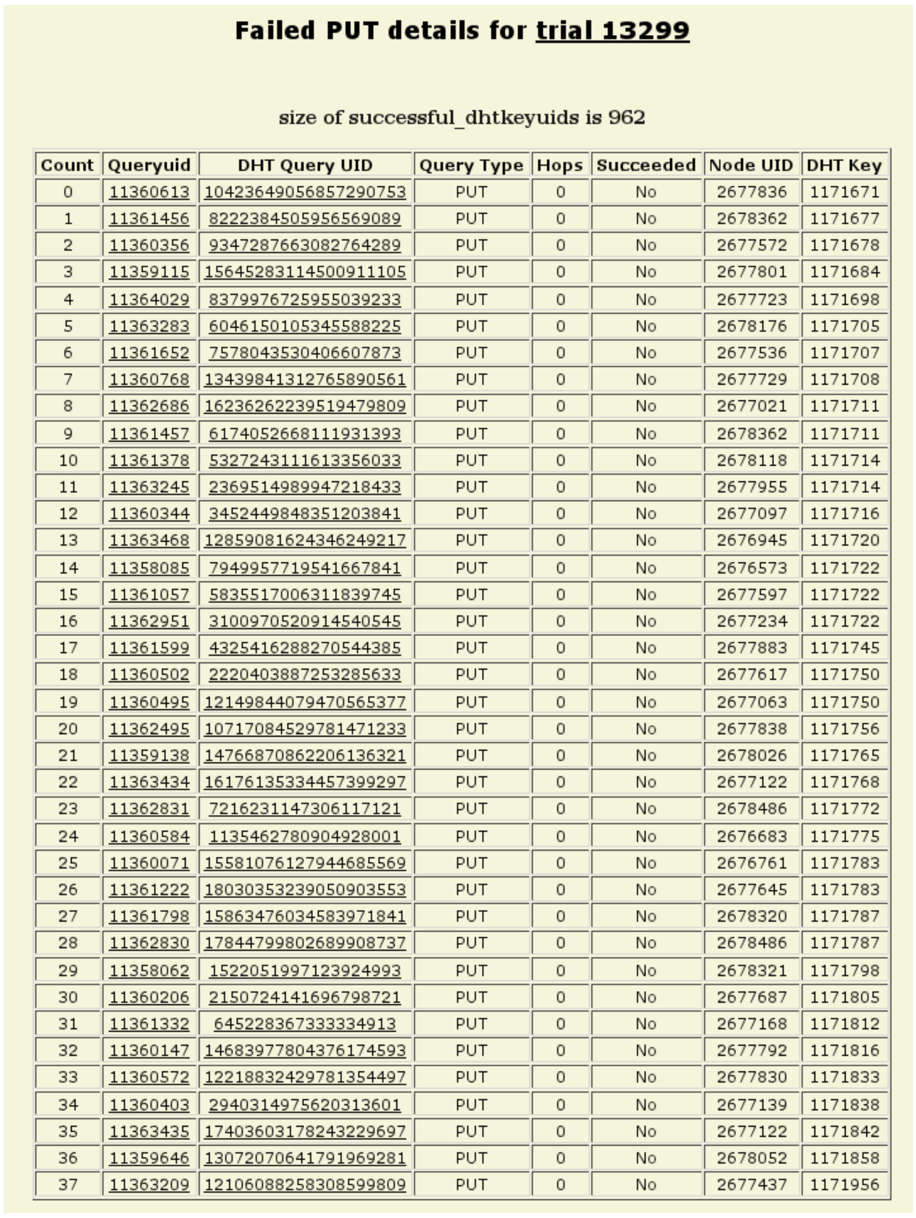
Start Time	End Time	Messages	Succeeded	Percent Succeeded	Valid	Percent Valid Succeeded	Avg. Replicas This Round	Avg. Replicas Total	Avg. Hops	Max Hops	# Reached Closest
2010-12-23 06:03:27	2010-12-23 06:35:49	100	97	0.97	100	0.97	3.6	3.09	2.81	7	29
2010-12-23 06:35:49	2010-12-23 06:38:28	100	96	0.96	100	0.96	3.62	5.54	2.88	9	25
2010-12-23 06:38:28	2010-12-23 06:42:32	100	95	0.95	100	0.95	3.62	7.9	2.84	6	21
2010-12-23 06:42:32	2010-12-23 06:46:30	100	99	0.99	100	0.99	3.83	10.29	2.9	7	33
2010-12-23 06:46:30	2010-12-23 06:49:08	100	96	0.96	100	0.96	3.34	12.36	2.69	7	25
2010-12-23 06:49:08	2010-12-23 06:52:09	100	94	0.94	100	0.94	3.94	14.43	2.9	6	26
2010-12-23 06:52:09	2010-12-23 06:55:07	100	97	0.97	100	0.97	3.62	16.42	2.85	8	28
2010-12-23 06:55:07	2010-12-23 06:57:49	100	96	0.96	100	0.96	3.76	18.38	2.83	7	21
2010-12-23 06:57:49	2010-12-23 07:00:24	100	97	0.97	100	0.97	3.85	20.28	2.91	8	31
2010-12-23 07:00:24	2010-12-23 07:03:17	100	95	0.95	100	0.95	3.66	21.89	2.96	7	30

Fig. 6.4: The web page used for viewing in depth trial information.

Figure 6.5 which displays each of the failed *PUT* requests. Finally, clicking the "Queryuid" of the request provides all of the hop-by-hop routing information collected.

Figure 6.6 displays the routing information for one particular failed *PUT* request. Here, detailed information is given about the path through the network that the request traveled. This information is displayed both as a table of all the routing information that was logged about the request as well as graphically for all queries for this particular key. The closest node(s) to the key in the network at the time are shown as well for additional troubleshooting. For instance, if the closest node to a key for a *PUT* request does not store the data, this could indicate a problem with the code for performing distance metric computation. Each node is identified by a specific unique identifier, which is the number in the center of the node before the colon. GNUnet's DHT uses a distance metric to determine the distance between a peer's identifier and the request key; the result of this metric between the node and the request key is the number after the colon following the peer's unique identifier.

In this case we do not see any red (self looping) edges, which means the *PUT* message was discarded by the GNUnet peer, or packet loss was experienced on the network. This could be the result of too much traffic on the outgoing or incoming queues or some other network failure. The graph would display any node which stored the data in blue, so we also see that



Failed PUT details for trial 13299

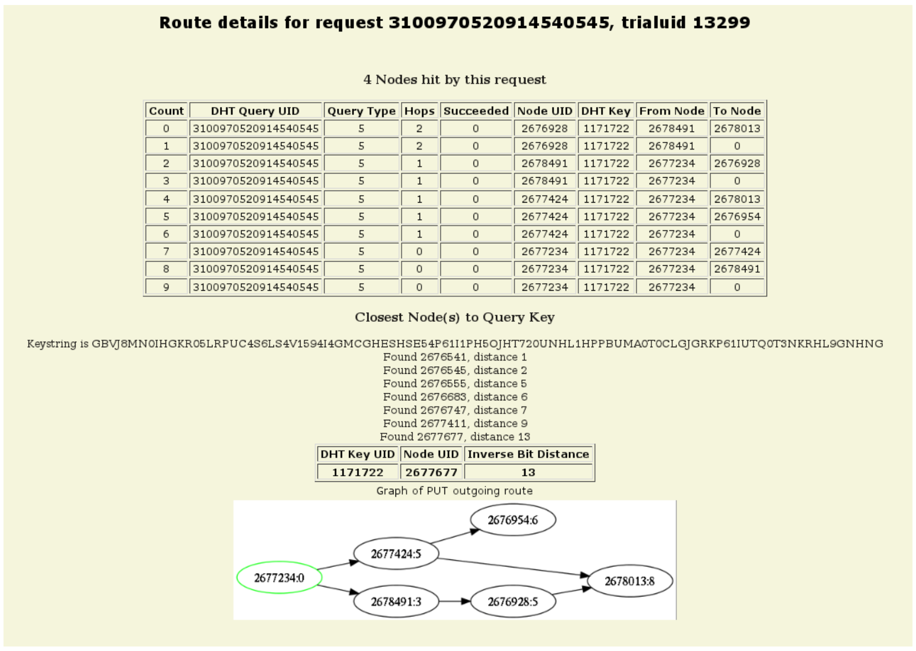
size of successful_dhtkeyuids is 962

Count	Queryuid	DHT Query UID	Query Type	Hops	Succeeded	Node UID	DHT Key
0	11360613	10423649056857290753	PUT	0	No	2677836	1171671
1	11361456	8222384505956569089	PUT	0	No	2678362	1171677
2	11360356	9347287663082764289	PUT	0	No	2677572	1171678
3	11359115	15645283114500911105	PUT	0	No	2677801	1171684
4	11364029	8379976725955039233	PUT	0	No	2677723	1171698
5	11363283	6046150105345588225	PUT	0	No	2678176	1171705
6	11361652	7578043530406607873	PUT	0	No	2677536	1171707
7	11360768	13439841312765890561	PUT	0	No	2677729	1171708
8	11362686	16236262239519479809	PUT	0	No	2677021	1171711
9	11361457	6174052668111931393	PUT	0	No	2678362	1171711
10	11361378	5327243111613356033	PUT	0	No	2678118	1171714
11	11363245	2369514989947218433	PUT	0	No	2677955	1171714
12	11360344	3452449848351203841	PUT	0	No	2677097	1171716
13	11363468	12859081624346249217	PUT	0	No	2676945	1171720
14	11358085	7949957719541667841	PUT	0	No	2676573	1171722
15	11361057	5835517006311839745	PUT	0	No	2677597	1171722
16	11362951	3100970520914540545	PUT	0	No	2677234	1171722
17	11361599	4325416288270544385	PUT	0	No	2677883	1171745
18	11360502	2220403887253285633	PUT	0	No	2677617	1171750
19	11360495	12149844079470565377	PUT	0	No	2677063	1171750
20	11362495	10717084529781471233	PUT	0	No	2677838	1171756
21	11359138	14766870862206136321	PUT	0	No	2678026	1171765
22	11363434	16176135334457399297	PUT	0	No	2677122	1171768
23	11362831	7216231147306117121	PUT	0	No	2678486	1171772
24	11360584	1135462780904928001	PUT	0	No	2676683	1171775
25	11360071	15581076127944685569	PUT	0	No	2676761	1171783
26	11361222	18030353239050903553	PUT	0	No	2677645	1171783
27	11361798	15863476034583971841	PUT	0	No	2678320	1171787
28	11362830	17844799802689908737	PUT	0	No	2678486	1171787
29	11358062	1522051997123924993	PUT	0	No	2678321	1171798
30	11360206	2150724141696798721	PUT	0	No	2677687	1171805
31	11361332	645228367333334913	PUT	0	No	2677168	1171812
32	11360147	14683977804376174593	PUT	0	No	2677792	1171816
33	11360572	12218832429781354497	PUT	0	No	2677830	1171833
34	11360403	2940314975620313601	PUT	0	No	2677139	1171838
35	11363435	17403603178243229697	PUT	0	No	2677122	1171842
36	11359646	13072070641791969281	PUT	0	No	2678052	1171858
37	11363209	12106088258308599809	PUT	0	No	2677437	1171956

Fig. 6.5: The web page used for viewing specific query information.

this data was never stored in the network.

Figure 6.6 showed only the outgoing part of a request because the initial *PUT* request failed. Figure 6.7 shows what is displayed for a successful *GET* request. Since the request was successful this means that at some point a *PUT*, *GET*, and *GET REPLY* were all initiated and succeeded for the particular key in question. All the pertinent information for all of these queries is shown in Figure 6.7. Nodes displayed in green indicate those



Route details for request 3100970520914540545, trialuid 13299

4 Nodes hit by this request

Count	DHT Query UID	Query Type	Hops	Succeeded	Node UID	DHT Key	From Node	To Node
0	3100970520914540545	5	2	0	2676928	1171722	2678491	2678013
1	3100970520914540545	5	2	0	2676928	1171722	2678491	0
2	3100970520914540545	5	1	0	2678491	1171722	2677234	2676928
3	3100970520914540545	5	1	0	2678491	1171722	2677234	0
4	3100970520914540545	5	1	0	2677424	1171722	2677234	2678013
5	3100970520914540545	5	1	0	2677424	1171722	2677234	2676954
6	3100970520914540545	5	1	0	2677424	1171722	2677234	0
7	3100970520914540545	5	0	0	2677234	1171722	2677234	2677424
8	3100970520914540545	5	0	0	2677234	1171722	2677234	2678491
9	3100970520914540545	5	0	0	2677234	1171722	2677234	0

Closest Node(s) to Query Key

```
Keystring is GBVJ8MN0IHGKR05LRPUC4S6LS4V1594I4GMCGHESHSE54P61I1PH5OJHT720UNHL1HPPBUMA0T0CLGJGRKP61IUTQ0T3NKRHL9GNHNG
Found 2676541, distance 1
Found 2676545, distance 2
Found 2676555, distance 5
Found 2676683, distance 6
Found 2676747, distance 7
Found 2677411, distance 9
Found 2677677, distance 13
```

DHT Key UID	Node UID	Inverse Bit Distance
1171722	2677677	13

Graph of PUT outgoing route

Fig. 6.6: The full web view for specific route information, showing the route of a *PUT* request. Because there are no red (self routed) edges which would indicate a request termination, and no blue vertices indicating request success, message loss was experienced on this route.

that initiated the request, and those in blue are nodes at which the request successfully terminated.

These tools for scheduling and displaying trial and route information have been invaluable in tuning our routing algorithm, and fixing implementation bugs. We have also implemented trial aggregation, which enables categories of trials, and functionality so that multiple trials (or groups of trials) can be selected and compared side by side. Finally, we also provide the ability to compute, graph and display standard deviations of appropriate metrics for trials which are repeated multiple times. While this code isn't intended for public consumption it serves as an easy way for us to produce graphs and data in a clean way on the fly for use in publications.

6.8.6 Database Back-end

We chose to use the MariaDB database [55] as our database back-end because of its widespread adoption, ease of use and plethora of client libraries to communicate with the database. We are not storing data requiring complex structures or stored procedures, and have found that with the proper indexes most operations return fast enough for our uses thus far. However, there is no requirement of a specific database; to use a different one

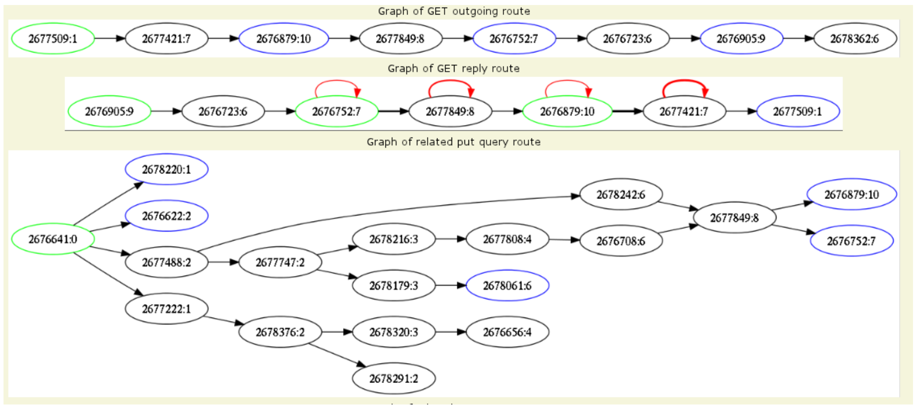


Fig. 6.7: The web page used for viewing specific route information. This example shows the path of an outgoing *GET* request accompanied by the route that the reply took from the responding peer back to the initiator, and the path of the original *PUT* request where the data was stored in the network. The red (self routed) edges on the *GET* reply route show instances where duplicate replies were detected, so the forwarding peers on the response path could safely discard these messages to save bandwidth.

would only require changes to the DHT logging subsystem in GNUnet, and dropping in a different connector for the web files and processing scripts.

6.8.7 Database Interaction (Data export)

As with the logging options discussed in Section 6.8, we have multiple ways for the DHT profiler and GNUnet peers to insert data into the database. Specifically, there are three different plugins for the DHT logging library which are loaded based on the specified configuration. The first plugin connects directly to a running database server, the second outputs full SQL commands to text files and the third outputs raw data directly to file. While the last plugin results in the fastest database interaction, it also requires some post processing before the data can be loaded in the database.

The direct connect plugin requires a live database which every peer is able to connect to. Each peer connects to the database server on startup and records are inserted immediately whenever the the logging API is called. This plugin operates in the most straightforward fashion; as an example, on startup each peer inserts its own peer identifier into the database. The database returns a unique identifier which is the internal database representation of that peer. Subsequent insertions into the database need only use this unique identifier, and not transmit the entire peer identity each time to identify the peer. Similarly, peers can query the database for information about which trial is currently being run, which round in the trial it is and runtime parameters. For these reasons this plugin would be preferred;

however when running trials with more than around 250 peers the database server becomes a bottleneck, and messages being sent between peers and the server slow down the performance of the entire trial.

The second plugin was created to ease the burden of connecting (typically over a network link) to a database server for each running peer. Instead, this plugin simply writes out the SQL queries the direct connect plugin would have sent to the server to a single file per peer which can be copied to the server and directly imported. While at first this method seemed quite simple, there are some obvious problems with it. To begin with, we had to do some post-processing of the files to make sure that order was preserved. Also, this method was quickly revealed to be painfully slow. At the outset there were SELECT statements intermixed with INSERT statements which retrieved unique identifiers for peers and keys. We then modified the order of insertion so that all peer identifiers were inserted first, then keys, etc. We then replaced any occurrence of those identifiers with the unique ids from the database. Locking tables and formulating queries appropriately helped us improve the timing of this method as well, but as we began emulating more and more peers in the network the size of the text files, the pre-processing overhead and the duration of data insertion led us to seek another method.

The third and final plugin we created uses a combination of the second plugin (for infrequent queries) and simply writes data values (excluding column information and more verbose SQL syntax) for frequent queries. One issue with this method is that it relies on a specific file format for MariaDB/MySQL, and is therefore not compatible with other SQL databases without extra manipulation. For this plugin, queries that are only executed a few times (once per trial, once per round, etc.) are still written out as SQL statements with one file per peer. For those queries that are repeated many times, each query type is written out to a unique file, where the data is simply written in a space and line delimited format. Upon termination of the trial, each of the specific query files from each peer are merged into one file, the normal SQL statements are executed and the aggregated files are imported using the LOAD DATA syntax available in MariaDB/MySQL. More pre-processing of these data files is required for this plugin. As with the previous plugin, the peer identifiers must be inserted first. After this step, the post processor then builds a mapping in memory of all peer identifiers the unique identifiers that were just created from insertion into the database. Next, all of the aggregated query data files must be searched and each instance of a peer identifier is replaced with the database unique identifier. The same process is then repeated for all keys used for requests in trials and the data is subsequently loaded into the database.

Figure 6.8 shows the stark difference between the second and third described SQL insertion methods. Originally the first method was adequate, but as we scaled our testing to greater numbers of peers it became clear that something better was needed. For much larger trials than those described in this thesis (i.e. greater than 10k peers), full logging even with the optimized SQL loading plugin likely takes too much disk space to be practical. Ideally, when running tests with large numbers of peers the minimal logging (*PUT*/*GET* success) should suffice.

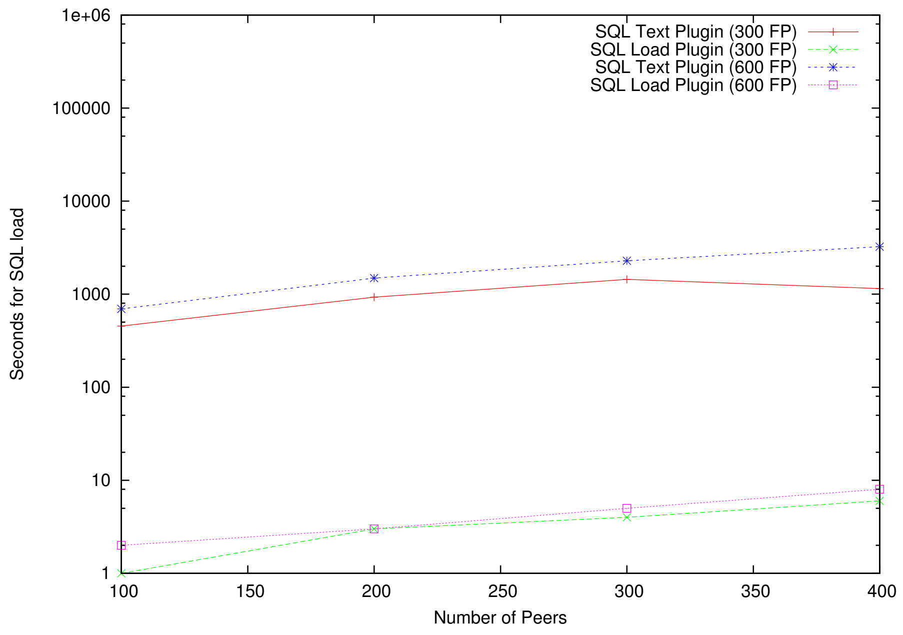


Fig. 6.8: Comparison of loading times between the raw SQL statement insertion method (which performs selects) and the plugin which writes raw data values to a file and the post processor inserts, bulk selects and LOADs data. These trials were run with varying numbers of peers and either 300 or 600 seconds during which peers issued *FIND PEER* requests (which created more data for insertion). Importantly, the y axis is plotted on a log scale, meaning the full text SQL insertion method takes between 2 and 3 *orders of magnitude* longer than the preprocessing and loading method.

6.9 Conclusion

In order to evaluate our routing algorithm, including the full implementation down to the network layer, we have created the emulation framework described in this chapter. This framework allows the running of variable numbers of peers in different underlay and overlay topologies up to a large scale. We have also developed a DHT specific profiling driver to perform DHT operations on the resultant network. This framework includes the ability to modify peers during runtime, such as setting certain peers to perform malicious actions. We learned many things from this implementation, and have detailed those important experiences as well in Section 6.6.

To support multiple levels of logging details from our specific DHT testing we implemented a number of different methods for data extraction; including storage in a database and numerical statistics. Finally, we also created a web based system for scheduling trials to run and for viewing the results of those trials.

In the next chapter, we present the design for R^5N; our new secure DHT routing algorithm. After examining the design in detail and evaluating it mathematically, we then provide detailed results from our evaluation of R^5N obtained using the framework presented in this chapter.

7. R^5N– RANDOMIZED RECURSIVE ROUTING FOR RESTRICTED ROUTE NETWORKS

This chapter details our implementation of R^5N, or “Randomized Recursive Routing for Restricted Route Networks”. This chapter brings together the insight gained from research on other P2P networks detailed in Chapters 2 and 4, and utilizing other system components as outlined in Chapters 3 and 5. We show results from a wide range of scenarios, from the best case underlay topologies required by Kademlia to restricted-route topologies that R^5N was designed for. The presented results also demonstrate the scalability of the emulation framework we implemented and described in Chapter 6. The overall results show that R^5N performs as expected in most cases, scales well and handles malicious participants either on par with, or much better than, a recursive implementation of the original Kademlia routing algorithm. Portions of the text and data that make up this chapter will be published at NSS 2011 [56]. Following this chapter, we conclude the thesis with a summary of the relevance of this work as well as some possible directions for future work on the design and implementation.

7.1 Introduction

Distributed Hash Tables (DHTs) [143, 153, 173] are a key data structure for the construction of completely decentralized applications. DHTs are important because they generally provide a robust and efficient means to distribute the storage and retrieval of key-value pairs.

In recent years, DHT designs have become increasingly efficient and robust under churn [102, 130, 147, 175] and Sybil attacks [47, 103, 170, 185]. Other research has addressed implementation concerns, such as optimizing network performance. In practice, modern DHTs are often not deployed over an entire P2P network and are instead limited in scope to so-called super-nodes. The primary reason for this is that virtually all previous DHT routing algorithms (with the notable exception of Freenet [159]) are based on the fundamental assumption of universal connectivity between all participating nodes.

This assumption means that modern DHTs cannot function properly in ad-hoc wireless-, sensor-, friend-to-friend- or other networks with limited connectivity. Following [159], we refer to such networks where peers are not free to directly connect to arbitrary other peers (and therefore route in the DHT) as *restricted-route* networks. We need to distinguish between the network topology created by the peer-to-peer overlay and the underlying network infrastructure, so we also use the term restricted-route underlay topology to refer to such limitations imposed on the overlay routing algorithm.

This chapter motivates and describes our new randomized DHT routing algorithm, which enables the GNUnet DHT to operate effectively over restricted-route networks and also increases security and resilience to various attacks compared to existing algorithms. R^5N only assumes that the topology is connected and, in particular, does not require or use a coordinate system for organizing peers. A primary goal of R^5N is providing an open network where users can join or leave at any time without approval by a certificate authority or any other trusted entity.

The high-level R^5N design itself is deceptively simple, essentially combining a random walk with recursive Kademlia-style [112] routing. Our design also includes topology augmentation using the combination of distance vector and onion routing (described in Chapter 5), a novel replication strategy and an API to verify content integrity. Using distributed emulation, we demonstrate that this new algorithm has performance comparable to Kademlia if the underlay is unrestricted, and outperforms Kademlia and random walks for various restricted-route topologies. We also show that our algorithm has advantages in terms of availability and fault-tolerance, especially in the presence of malicious participants. Compared to Kademlia, we generally see a larger number of replicas and higher success rates for data retrieval. Our algorithm has been created and deployed using the GNUnet P2P framework, which also includes a free software implementation of the measurement tools and topology generators used for the experiments presented in this thesis.

The remainder of this chapter is structured as follows. Section 7.2 defines restricted-route topologies and explains known techniques for overlay construction and routing in P2P networks. Section 7.3 describes our routing algorithm in detail as well as important parameters for replication and routing. Section 7.4 provides a mathematical comparison of random routing, R^5N and the Kademlia routing algorithm. Section 7.5 provides justification for our choice of the randomization parameter used in R^5N, based on applications of Markov mixing theory. Section 7.6 outlines our implementation and presents experimental results from some small scale testing to verify our implementation and show that it works as expected under various conditions. Section 7.7 then extends the interesting results discovered from small scale testing to larger scale tests approaching real world network sizes.

7.2 Related Work

A DHT imposes structure upon a network underlay by connecting peers to a certain subset of all nodes in the network. The size and method of construction of the routing table is one of the key design choices that distinguish DHTs. For example, Kademlia [112] has routing tables of size $O(\log n)$ and can route requests to the proper destination with $O(\log n)$ steps.

Another key design choice for a DHT is the routing or lookup behavior, which is categorized either as iterative or recursive [75]. Using iterative routing, the initiator directly connects to each hop and retrieves information about the next hop until the initiator has a direct connection to the final destination. As a result, the initiator of a request has full control over which node(s) the request is forwarded to at each step — and can possibly tackle problems (such as node failures or malicious participants) during request propagation (for example, by choosing alternative paths).

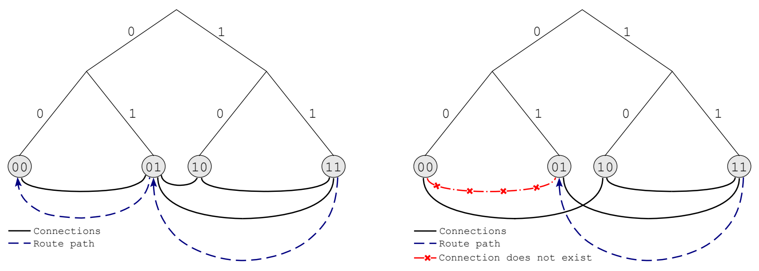
With recursive routing, requests are forwarded through the network from the first hop onwards according to the routing algorithm and the initiator is only involved again as the final destination of the response, if there is any. Recursive routing is generally faster than iterative routing since fewer connections need to be established and significantly fewer round-trip times are required. Another key benefit of recursive routing is that the initiator does not have to be able to connect to each peer that participates in routing. However, recursive routing is also less fault-tolerant due to the initiator's lack of control.

7.2.1 Kademlia

We use a modified version of Kademlia [112] as the basis of our routing algorithm. The Kademlia algorithm has been shown to work well in networks with common rates of churn [130], and in practice has proven capable of handling millions of peers [171]. Kademlia uses XOR to determine the distance between elements in the key space. This means that, given two nodes a and b in the network, the *distance* between a and b is given by $D_{a,b} = L(a)$ XOR $L(b)$ where $L(x)$ is the identifier of peer x. A key advantage of the XOR metric is that routing is symmetric; links between peers will be of mutual value and both directions are expected to see equal utilization.

Kademlia's routing table is structured as an array of k-buckets. Kademlia uses as many k-buckets as there are bits in the address space. Each k-bucket can hold up to k peers. The i-th k-bucket stores up to k peers whose identifiers are between distance 2^i and 2^{i+1} from the local peer. Routing in Kademlia is iterative; at each hop the initiating node picks r closest peers for the next hop. Those r peers are queried and return a set of peers closer to the key, and routing continues in this fashion until no closer peers are found. Finally, Kademlia stores data at the r closest peers to the key.

Kademlia achieves $O(\log n)$ routing performance: at each hop the distance to the destination is at least halved (Figure 7.1a).



(a) Normal Kademlia **(b) Restricted Kademlia**

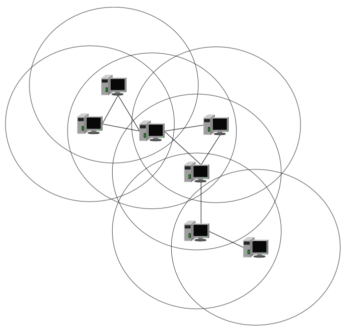
Fig. 7.1: Illustration of Kademlia routing for key "00". In (a), routing in a four-node Kademlia network ($r = 1$) with a set of connections that satisfies Kademlia's requirements. In this topology, all requests for keys starting with "00" terminate at node "00". In (b), the same requests are routed, but a single link required by Kademlia is unavailable. Here, requests started at peers "10" and "00" are routed to the correct node ("00") while those started at "01" or "11" are incorrectly routed to "01".

One failing of Kademlia is that it has been shown vulnerable to numerous attacks, such as poisoning [103] and Sybil [170] attacks. For example, an adversary may want to deny participants access to a particular key. This can be achieved by creating r peers with identifiers closer than the closest current peer to the key; afterwards, all requests will effectively end at an adversary-controlled peer. Access to the data is then under the control of the adversary.

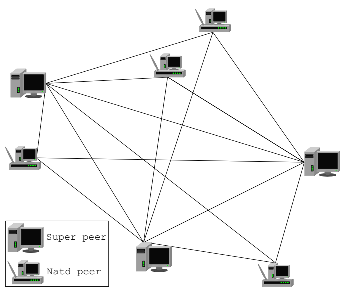
7.2.2 Restricted-Route Topologies

We use the term "restricted-route topology" to refer to a connected underlay topology which prohibits (restricts) direct connections between certain nodes. More formally, given a connected graph $G = (V, E)$ where V is the set of vertices and E the set of edges, the graph represents a restricted-route topology if $|E| < |V|^2$. Common examples for restricted-route networks are given in Figure 7.2.

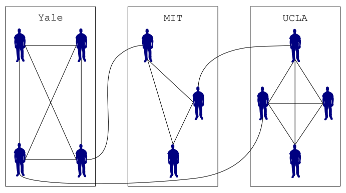
Common DHT routing algorithms show diminished performance or even arrant failure when operating over a restricted-route underlay. Restricting DHTs to "super-peers", those peers that are able to route without such restrictions, does not work for physical networks or friend-to-friend networks and may not work well on the Internet in the future as more and more restrictions are imposed on users, reducing the fraction of peers that could function as super-peers.



(a) Wireless Mesh



(b) Super Peer/NATed



(c) Social

(d) Physical

Fig. 7.2: Examples for sparse, restricted-route networks. (7.2a) Wireless networks are often sparse restricted-route networks due to limited signal strength. (7.2b) Firewall rules, virtual private networks and private subnets created using network address translation (NAT) are the reason why most P2P networks on the Internet today have to deal with a restricted-route topology. (7.2c) Friend-to-Friend P2P networks, constructed from social networks in order to enhance security [185], are also inherently restricted-route topologies. (7.2d) The physical underpinnings of the Internet also form a restricted-route network (image from chrisharrison.net).

7.2.3 T-DHT

T-DHT [98] is a DHT routing algorithm designed specifically for ad-hoc wireless networks, with a focus on wireless sensor networks for their evaluation. In T-DHT, peers in the network use a topology discovery mechanism to determine how distant they are in the network from so-called "reference nodes". Based on the number of overlay hops from the reference nodes each peer is assigned a coordinate in a global coordinate system. From this construction, nodes which are "close" based on the number of hops between them are also "close" in the virtual coordinate system. This allows efficient routing by greedily forwarding requests toward the virtual coordinate.

While T-DHT provides efficient routing in ad-hoc wireless networks, it has the major failing of requiring a global coordinate system that all peers in the network agree on. This type of global information scheme requires global knowledge of peers about the underlying topology of the network, including the total number of nodes in the network. Also, peers joining the network must be given a location in the global coordinate space based on information from other peers in the network, which may or may not be trustworthy. Therefore, peers do not make routing decisions based solely on local knowledge, but must in a sense trust the other peers in the network.

7.2.4 Freenet

Freenet [159][1] is the only efficient DHT design that we are aware of which works well in restricted-route networks without coordinates. The Freenet routing algorithm was created for so called "darknets" which are created by building a topology out of a real world social graph. These social graphs have been shown to be Small-World graphs [38,185], where short paths exist between peers even though the topologies are sparse.

Freenet uses location swapping between peers in order to structure the overlay topology, and is able to route in $O(\log n)$ steps. The main problem with Freenet's DHT is the inherent vulnerability of the critical location swapping operation. This operation allows an adversary with only a few peers located anywhere in the network to cause massive peer identifier clustering, leading to possible data loss and destroying the load balancing properties of the DHT. Furthermore, natural churn can cause similar problems even in the absence of malicious peers. We are not aware of any good solution to these problems for an open network, making Freenet's location swapping approach unusable for the creation of a secure DHT for restricted-route underlays.

Additionally, while the original intention of Freenet was to rely on trusted connections between peers, building a network in this way proved ineffective in the real world. This is because small P2P networks, such as Freenet, typically have a diverse set of users from around the world. However, these users are unlikely to know each other in the real world, so the Freenet darknet could not be bootstrapped without allowing connections from untrusted peers. The result of this is that Freenet operates for most users in an "opennet" mode; where the supposed trusted connections are made with anonymous peers via a discovery mechanism. Thus, the design of Freenet is vulnerable to malicious peers and churn, but even achieving a Small-World topology in the real world is problematic. The issues we encountered and attacks performed on Freenet were presented in detail in Chapter 2.

7.2.5 Randomized Designs

One interesting subcategory of restricted-route networks are sparsely connected networks, where $|E| \sim |V| \cdot \log |V|$. These can often be modeled as random graphs, and it has been shown that flooding [106] or random walks [5] are the method of choice for searching such random graphs with

[1] This is not the original Freenet design from 1999 but their approach from 2006.

no structure; though these unstructured techniques are costly ($O(\log n\sqrt{n})$) compared to modern structured routing algorithms [64].

Lv et. al provide a definitive overview of random walk methods for searching unstructured P2P networks in [106]. They show, using simulation and proofs, that random walks are as good as flooding for search in such networks, and require significantly fewer hops. However, their method requires $\sqrt{n}$ average replicas in the network and significantly more hops than our randomized search. Also, the topologies used for their work were very sparsely connected (a total of four topologies of approximately 10k nodes, all of which had average node degree less than five). Our research covers a wider and more realistic range of topologies, and our approach achieves similar success rates with fewer total hops.

A closely related randomized design for unstructured networks is [64], which presents a recursive random walk routing algorithm aimed at obtaining a random sample of the network with each request. Assuming there are enough replicas present in the network, the birthday paradox guarantees that requests will succeed with high probability. The main differences between [64] and [106] lie in replication and next-hop strategies. Whereas the latter relies on completely randomized next-hop selection, the former chooses peers based on the number of edges of the possible next-hop neighbors. While this strategy makes the search optimally random, it relies on non-local information to make routing decisions, and may be susceptible to attack. The other difference between these two algorithms is that [64] adjusts replication based on the estimated number of copies currently in the network, where [106] always replicates at any peer encountered along the query path. The result of this is that [64] does a better job finding less popular items in the network. Again, [64] evaluated only three random topologies, and while comparable to the heuristic approach taken by [106], still requires many more hops to find data than R^5N.

While both of the previous approaches are similar to R^5N in the first stage of routing, they are meant for completely unstructured networks and do not take into account the restricted-route topologies addressed by R^5N. Finally, both of the research papers on the previous routing algorithms designs go into detail about replication and handling peers leaving the network due to normal churn scenarios. They do not address problems due to malicious participants in the network, and while some of these (such as Sybil attacks) are mitigated to some degree by unstructured topologies, other attacks may still be possible.

Bubblestorm [178] provides a probabilistic search in unstructured random networks. The approach taken by Bubblestorm is to replicate queries and data within a certain "bubble"; as long as the bubbles of the querying peers and the peers with the result intersect the search will succeed. This is similar to the approach taken by R^5N, but with a reliance on completely random topologies. Bubblestorm requires precise knowledge of each peer in the overall topology and network size. Bubblestorm employs a “weight” which is similar to our replication factor r and avoids forwarding requests to peers encountered previously.[2] Bubblestorm is intended to deal with large scale network failure, simulating up to 90 percent total system failure with minimal recovery time. This is achieved by combining message flooding with random walks. Thus, their approach is not concerned with efficient routing.

Spinneret [150] discusses a method for building a structured overlay out of an unstructured topology with some relaxed constraints on that structure. Specifically, a Chord like ring topology is created so peers have connections at given distances from themselves on the ring, but those peers are chosen from a range instead of those at a specific distance. They then use both random walks and greedy routing (possibly simultaneously) so that queries can succeed even in topologies with high churn or poor connectivity. The idea of combining both types of search is similar to R^5N although universal connectivity is still required by Spinneret. Spinneret requires two distinct routing algorithms,has no protection against malicious peers, and leaves the problem of replication up to application using the DHT.

NoN (Neighbor of Neighbor) routing was introduced in [108], and shows that in Small-World and other randomized networks used by DHTs the average number of hops can be reduced from $O(\log n)$ to $O(\frac{\log n}{\log \log n})$. This is accomplished by routing at each hop not to the immediate nearest peer (greedy routing) but by maintaining knowledge of the immediate peers neighbor's *neighbors*. Each query is then routed to the immediate neighbor which has the closest neighbor to a query. Maintaining the list of each neighbor's neighbors greatly increases the size of the routing table (or requires querying each neighbor for each hop); this result is interesting because R^5N achieves similar functionality with the distance vector transport plugin.

Various so-called “randomized” DHT designs have recently been proposed [18, 36, 107, 189] to deal with problems associated with the deterministic nature of traditional DHTs. For instance, Symphony [107, 189] orders nodes in a ring and randomly chooses peers via random peer sampling (RPS) to keep in the routing table based on distance (similar to Chord). This enables routing with fewer long distance links than previous designs and increased flexibility in terms of specific connections, resulting in reduced overheads. Similarly, D2HT [18] assumes that a Small-World topology can be created in the overlay using neighbor gossip to discover and select nearby and long range links. While these and other designs reduce the complexity of maintenance and preserve efficiency, they still rely on unrestricted underlay topologies.

Danezis et. al [36] presented a semi-randomized version of the Chord

[2] We achieve this in R^5N using a Bloom filter; Bubblestorm manages this by remembering queries locally (to prevent forwarding to the same peer twice), and relies on the randomness of the topology to prevent “large” loops.

DHT. Their design, which was created to foil Sybil attacks on the DHT, altered Chord to use multiple possible strategies when selecting the next hop for a query. While the default strategy is to always greedily forward the query to the next closest node (closeness strategy), the authors implemented others including diversity (spread queries equally amongst peers) and zigzag (combine closeness and diversity). The result was that randomization of query paths helped resist Sybil attacks.

In contrast to R^5N, [36] requires iterative routing and full routing table transfer betweens peers at *every* hop. This makes the routing overheads in their design significantly higher than most previous DHT designs [112, 143, 173] as well as R^5N. While our approach provides similar protection, we focus on a more diverse range of malicious participants and, most importantly, on restricted-route topologies which necessitate recursive routing. Another key difference is that [36] alternates between strategies at every hop, whereas ours is more "focused" towards the end of the route (making the possibility of finding a nearest peer more likely).

These randomized designs do help to solve specific real world problems and improve resilience to high rates of churn and malicious peers. However, to the best of our knowledge, none of these designs actually cope well with restricted-route underlay topologies, further motivating the randomized design such as R^5N.

7.3 Design of R^5N

The basic idea of R^5N is to take advantage of the limited connectivity of restricted-route networks by using the large number of peers that are closer to a key than any of their neighbors for replication. A PUT operation is used to store data at a random subset of these peers, and GET requests then attempt to reach one of the replicas. PUT requests are repeated at a certain frequency to refresh data, combat churn and increase replication. Since R^5N performs non-deterministic routing, repeated PUT requests are likely to result in data being stored at different peers. Furthermore, since the refresh period is significantly shorter than the timeout of content at the replica nodes, the chance of success for subsequent GET operations is increased.

Despite this replication, a GET request may still fail to find the target value. In this case, R^5N expects peers performing GET requests to retry a few times. Since GET requests are also non-deterministic, repeating the operation has a high chance of reaching different peers and hence improves the chance of finding the data. While the design guarantees that the chance of failing to find existing data declines over time, the specifics depend on the replication parameter r, the network topology and the number and behavior of adversaries in the network.

Since both GET and PUT requests take different paths each time, an adversary has little chance to successfully place his nodes in the network to block particular key-value pairs. Depending on how the restricted-route underlay is constructed, an isolation attack on nodes may still succeed.

The remainder of this section will detail the various components required for the R^5N routing algorithm. Specifically, we will discuss routing table construction, the use of bounded onion-routing to augment the underlay, routing, replication and application-level requirements (specifically content validation).

7.3.1 The Routing Table

Routing tables in R^5N are closely modeled after Kademlia routing tables, and are constructed and maintained in a similar manner. The distance function is similar Kademlia's XOR metric:

Definition 7.1 (Proximity to Identifier). *Given two identifiers l, m, the proximity $D(l,m)$ between l and m is $x := D(l,m)$ if and only if:*

$$\bigvee_{b \in [0,x)} l_b = m_b \qquad \texttt{and} \qquad l_x \neq m_x. \tag{7.1}$$

This simply defines the *distance* between any two identifiers to be the number of consecutive matching high-order (leftmost) bits. Clearly, the inverse is also true, so $D(l,m) = D(m,l)$.

The routing table is made up of k-buckets with one bucket for each bit in the identifier space (our GNUnet implementation uses 512-bit identifiers). Each of these buckets can hold k entries for peers with matching prefix length equal to the index of the entry in the routing table.

Definition 7.2 (Routing Table). *Let $R_p(n)$ be the n-th bucket of the routing table of peer p. If two peers with identifiers i and j are connected, then $j \in R_i(X(i,j))$ and $i \in R_j(X(j,i))$ (unless the respective bucket has more than k entries, in which case the oldest k connections are used).*

When a peer joins the network, it sends a *FIND PEER* request out to some known peer(s) already in the network. The request is routed through the network and peers along the path may connect to the new peer (depending on the peers currently in their routing tables). When the request reaches a nearest peer, it responds to the new peer with its full routing table. The new peer will continue to send additional *FIND PEER* messages until few new peers are found and then only perform *FIND PEER* operations at a lower frequency for maintenance.[3] As in Kademlia [112], this approach results in $O(\log n)$ connections to neighbors (when in an unrestricted underlay topology) given n total peers in the network.

[3] Due to message size concerns, our implementation does not piggyback peer information on normal DHT route requests (as done by Kademlia).

7.3.2 Fisheye Distance Vector Underlay Augmentation

The above description of routing table construction ignores the problem that peers will often not be able to communicate directly. In some topologies, this may simply result in a slightly reduced number of peers in the routing table. While this is enough to require randomized routing as described in the next section, a slight reduction in the number of peers in the routing table is not a problem for successful routing in general.

However, for sparse topologies, especially those with fewer than $O(\log^2 n)$ neighbors per node, the resulting routing table would be mostly empty (in relation to the size of the network). The resulting limited view of the network by the peer would make it difficult to make good routing decisions. R^5N addresses this problem by using a *distance-vector* layer below the DHT, described previously in Chapter 5. For Small-World underlays (restricted-route networks with a network diameter of $O(\log n)$), this layer provides the DHT with the ability to communicate with $O(\log^2 n)$ peers, even if the restricted-route underlay does not allow that many direct connections.

The distance vector layer uses a bounded variant of distance vector routing [136] to discover shortest paths to peers up to $D = 3$ hops away.[4] It can then be used to build tunnels, using layered encryption akin to onion-routing [176], to communicate non-anonymously with any of these peers.

The use of onion-routing makes it difficult for malicious neighbors to monitor or restrict connections to particular nodes. For Small-World networks with an average degree of $O(\log n)$, the choice of $D = 3$ will result in a large enough number of neighbors to sufficiently populate the R^5N routing tables of most peers. For more details on the specifics of the distance vector protocol that we have developed, see Chapter 5.

7.3.3 Routing

Routing in R^5N is performed in two distinct phases. In the first phase, a request for a key is routed for some number of hops in a random fashion (of course, next hop choices are limited to those connections in the routing table). In the second phase, routing is deterministic using the peers from the routing table that are closest to the given target. Each request includes the number of hops h that the request has traversed so far and each peer is supposed to increment the counter by one at each hop. If the hop counter exceeds a threshold of $T \approx \log n$ where n is the size of the network, the peer uses deterministic routing. The intuition behind this is that we first make the starting point in the network independent from the initiator and then find a nearest peer.

[4] The resulting topology is still a restricted-route topology in most cases. Making D large enough to eliminate problems of traditional DHTs with restricted-routes would cause performance and possibly security [58] problems.

We considered strategies that used a phase with a degree of bias in the neighbor selection instead of the binary choice between random and deterministic. For the underlay topologies we investigated, such strategies showed little advantage in terms of performance and have the disadvantage of being harder to analyze, understand and implement.

In addition to the hop counter, each request also contains a unique identifier and a Bloom filter which are used to prevent looping and to limit forwarding of the same request to the same peer repeatedly. The Bloom filter is updated with the list of peers selected for forwarding for the request at each hop, and those peers that match the bloom filter are excluded from the selection process henceforth.

7.3.4 Estimating Network Size

R^5N's two-phase routing requires a rough estimate of the size of the network in order to determine how many hops of random routing should be used before switching to deterministic routing. However, estimating the network size of P2P network is a notoriously difficult problem. If the network allows direct connections between most neighbors, the size of the network can be estimated directly from the routing table. If T is the average index of the first empty bucket in the routing table, then the size of the network is approximately 2^T and the the expected path length for Kademlia is also T. This and other similar simple methods do not work for restricted-route topologies. Other heuristic methods exist for estimating the size of a P2P network [114], and we intend to explore these methods in future work to discover which performs best for Small-World topologies.

For our presented results, we know the size of the network and therefore manually set the T parameter accurately. In a real world deployment, values of T between 3 and 6 cover network sizes between approximately 1,000 and 1,000,000 peers. Thus, even hard coding a value in this small range should work for most real world networks, with the obvious caveat that an overestimate will be slightly more costly and an underestimate may cause requests to be insufficiently randomized (and have higher failure rates).

7.3.5 Processing Requests and Replies

Each peer that receives a routing request performs the same basic sequence of operations. First, the peer determines whether it is closer to the key of the request than any of the peers in its routing table.

Definition 7.3 (Nearest Peer). *A peer i is considered nearest to random identifier l if for all $j \in R_i$, $X(j,l) \leq X(i,l)$.*

In other words, a peer is considered a nearest peer if the bit distance between its peer identifier i and the key k is greater or equal to the bit distance between any of the connections of i and k. For a random network, we can easily calculate the expected number of nearest peers.

Remark 7.4 (Number of Nearest Peers). *For a random network with n peers and α random connections per peer, the expected number of nearest peers in the network to any random key is $\frac{n}{\alpha+1}$.*

This calculation is correct for completely random topologies (e.g. Erdős-Rényi and Small-World) without structure, but the numbers are skewed by routing table restrictions.This is because sending *FIND PEER* requests adds peers to the routing tables preferentially based on location. Therefore, Table 7.1 lists the number of nearest peers for some of the different *actual* topologies we used.

If the current peer is a nearest peer (and $h \geq T$), it is possible that the request is not forwarded at all, depending on the type of request (for instance, *PUT* requests are not forwarded past a nearest peer, but *GET* requests that might have multiple results may still be forwarded).

Peers will cease to forward a message if the hop count exceeds an upper bound of $2 \cdot T$. Peers also stop forwarding requests for a neighbor if the *GET* queue for that neighbor is full. Each time a *GET* request is processed for a neighbor, it is added to the *GET* queue associated with that neighbor. A timeout is used to remove entries from the queue. This limits the number of outstanding *GET* requests that will be handled for a particular peer. If the request is forwarded to other peers in the network, the number of forward replicas is calculated according to the replication level, network size estimate and number of hops traversed as described in Section 7.3.6.

For handling replies, a bounded number of active request keys and the respective identity of the preceding peer is maintained. Responses are forwarded along the request path in reverse until they reach the initiating peer or are discarded by a peer that lacks path information, for example due to memory limitations. It should be noted that most other DHTs do not require this additional state since, in traditional DHTs, the normal routing mechanism can also be used to route replies. For R^5N, this is not feasible due to path randomization. This randomization causes the success rate for replies reaching the intended initiator to be rather low. In contrast, randomization for the lookup is acceptable (as well as vital to our design!) since many peers are expected to store the data.

7.3.6 Replication

In R^5N, replication is used not only to protect against node failure, but also to improve the chances of a lookup operation finding the desired data in the absence of failures. For R^5N, the highest *GET* success rate would be achieved if there are $\frac{n}{c+1}$ replicas in the network (Lemma 7.4). We use

Allowed Topology	**Avg. Connections in Routing Table**	**Number Nearest**	$\frac{n}{c+1}$
2d-torus	4	580.09 ± 8.71	405
Clique	10.16	359.28 ± 26.92	184.10
Clique	12.32	276.64 ± 25.11	155.77
Clique	22.55	131.09 ± 17.39	92.04
Clique	26.49	110.82 ± 13.40	76.44
Clique	30.2	90.17 ± 11.73	67.05
Clique	32.62	85.94 ± 11.93	61.36
Clique	46.55	19.69 ± 1.62	42.59
Clique	50.01	2.57 ± 0.85	39.71
Clique	60.85	2.27 ± 1.77	32.74
Small-World	8.00	336.37 ± 12.71	225
Small-World	12.00	228.74 ± 10.12	168.75
Small-World	17.99	155.67 ± 7.31	106.57
Small-World	22.00	123.85 ± 9.66	88.04
Small-World	25.98	107.75 ± 6.12	75.06
Small-World	28.00	99.98 ± 6.35	69.83
Small-World	31.97	89.21 ± 5.49	61.42
Small-World	35.98	77.68 ± 5.64	54.76
Small-World	41.97	68.53 ± 4.86	47.13
InterNAT	9.63	715.60 ± 41.05	190.50
InterNAT	18.24	453.70 ± 31.40	105.25
InterNAT	22.09	371.07 ± 32.33	87.70
InterNAT	26.15	256.24 ± 27.12	74.59
InterNAT	29.66	200.38 ± 25.08	66.05
InterNAT	37.34	94.06 ± 13.24	52.82
Erdős-Rényi	7.85	319.49 ± 14.33	228.81
Erdős-Rényi	11.66	238.87 ± 11.09	160.71
Erdős-Rényi	13.57	203.64 ± 10.64	138.98
Erdős-Rényi	19.06	148.86 ± 7.91	100.95
Erdős-Rényi	27.86	99.38 ± 5.90	70.17
Erdős-Rényi	36.40	79.14 ± 6.12	54.14
Erdős-Rényi	44.42	65.19 ± 6.42	44.58
Erdős-Rényi	52.05	52.24 ± 5.61	38.17
Erdős-Rényi	59.43	48.88 ± 5.11	33.51
Erdős-Rényi	66.05	42.63 ± 4.01	30.20
Erdős-Rényi	72.93	39.48 ± 4.57	27.39

Tab. 7.1: Observed number of nearest peers for different topologies, all with $n = 2025$ peers.

$r \sim \sqrt{\frac{n}{c+1}}$; this choice represents a trade-off between the cost for *PUT* requests and the performance for *GET* requests.

If the initiator were to transmit r *PUT* requests to obtain r replicas, there would be a good chance of collision in the resulting paths and this might impose a strong burden on the direct neighbors of the initiator, especially since in the underlay the initiator may not even have r neighbors. Instead, R^5N attempts to have (on average) $1 + \frac{(r-1)h}{T}$ *PUT* requests active in the network at hop $h \leq T$.

Lemma 7.5. *Let h be the number of hops in the network that the request has already traversed. If the network is large enough that r random paths of length T are unlikely to merge and if $h < T$, than the average number of peers to which a peer forwards a request to should be*

$$\Upsilon_{r,h} := 1 + \frac{(r-1)}{T + (r-1)h} \tag{7.2}$$

in order to achieve the desired replication level r at T hops.

Proof of Lemma 7.5 by induction. To show: If each peer forwards to $\Upsilon_{r,h}$ peers at each hop from $h = 0$ to $h = T - 1$, at hop T, there will be at least r parallel paths.
We start by showing that at any hop h there are $\frac{T+(n+1)\cdot(r-1)}{T}$ paths.
Forwarding to $\Upsilon_{r,h}$ peers at each hop, the total number of paths at hop h can be found by the recursive formula τ_h, defined as follows:

$$\begin{aligned}
\tau_0 &= 1 + \left(\frac{r-1}{T}\right) \\
\tau_1 &= \tau_0 + \tau_0 \cdot \left(\frac{r-1}{T + 1 \cdot (r-1)}\right) \\
\tau_h &= \tau_{h-1} + \tau_{h-1} \cdot \left(\frac{r-1}{T + h \cdot (r-1)}\right)
\end{aligned}$$

We will next show that the recurrence relation can be simplified to:

$$\tau_h = \frac{T + (h+1) \cdot (r-1)}{T}.$$

Now, we show the inductive base case τ_0:

$$\begin{aligned}
\tau_0 = 1 + \left(\frac{r-1}{T}\right) &= \frac{T + (r-1)}{T} \\
&= \frac{T + (0+1) \cdot (r-1)}{T}
\end{aligned}$$

so τ_0, the base case for the inductive step, holds.

Next, assume τ_k is true, then:

$$\tau_k = \frac{T + (k+1) \cdot (r-1)}{T}.$$

Now,

$$\tau_{k+1} = \tau_k + \tau_k \cdot \left(\frac{r-1}{T+(k+1)\cdot(r-1)}\right)$$

Substituting τ_k for $\frac{T+(k+1)\cdot(r-1)}{T}$

$$\begin{aligned}\tau_{k+1} &= \left(\frac{T+(k+1)\cdot(r-1)}{T}\right)\\ &+ \left(\left(\frac{T+(k+1)\cdot(r-1)}{T}\right)\cdot\left(\frac{r-1}{T+(k+1)\cdot(r-1)}\right)\right)\\ &= \left(\frac{T+(k+1)\cdot(r-1)}{T}\right) + \left(\frac{r-1}{T}\right)\\ &= \frac{T+(k+1)\cdot(r-1)+(r-1)}{T}\\ &= \frac{T+(((k+1)+1)\cdot(r-1))}{T}\end{aligned}$$

Thus, τ_k implies τ_{k+1} which proves by induction that

$$\tau_h = \frac{T+(h+1)\cdot(r-1)}{T}$$

Our hop count h starts at 0, and $T \in \mathbb{N}$, so the hop counter has reached the total length T when $h = T - 1$.

Let $T = h + 1$, then:

$$\begin{aligned}&= \frac{T+(h+1)\cdot(r-1)}{T} = \frac{T+T\cdot(r-1)}{T}\\ &= 1 + r - 1 = r\end{aligned}$$

□

Clearly, $\Upsilon_{r,h}$ will generally not be a natural number. R^5N uses a biased random selection, forwarding to either $\lfloor\Upsilon_{r,h}\rfloor$ or $\lceil\Upsilon_{r,h}\rceil$ peers to reach on average $\Upsilon_{r,h}$ peers for the next hop.

7.3.7 Content Validation

A key concern for any DHT is the integrity of the content stored in the system. R^5N provides an application with hooks for integrity checks to detect malformed key-value pairs. Such pairs are then not forwarded or stored by well-behaved peers, which reduces storage and bandwidth requirements in the presence of faulty or malicious participants.

Another possible issue is allowing multiple values to be stored under the same key. Requests in R^5N include a Bloom filter [21] which matches replies already known to the requester. While Bloom filters offer a compact way to filter replies, they can also produce false positives. R^5N mitigates this problem by having the requester provide an additional mutation value which modifies the hash function used for testing the Bloom filter. By reissuing the request with a different mutation value, these false-positives can be eliminated with high probability.

Finally, R^5N also allows application-level hooks to specify that a particular *GET* request type only has a single possible value as a reply and hence forwarding is never required once an answer has been found. Applications can further use these hooks to verify self certifying data, such as Freenet CHK's [158]. These measures can allow those peers storing or forwarding data to verify that only valid data is being inserted.

We believe that providing such hooks to control content integrity and to allow early filtering of duplicate replies is crucial in order to provide an implementation of a general-purpose, secure DHT. Without such measures, adversaries can more easily pollute the DHT, consuming resources and harming availability. However, there are applications that do not require the overhead of these filtering mechanisms, which is why we allow the application to choose whether or not to use them.

7.3.8 Adversary Model

We consider a number of types of malicious adversaries with diverse goals in our design. We assume that an attacker has similar resources to other participants in the network, and may eavesdrop, alter and send and receive messages as a normal peer would. This adversary may also create multiple peers running simultaneously with free choice of peer identity for identification and lookup in the DHT. We also assume that adversaries may collude in order to achieve a specific goal. Finally, we assume that encrypted messages intercepted at the network level are unable to be decrypted by anyone other than the intended recipient.

7.3.8.1 Eavesdroppers

There are two kinds of adversaries that may eavesdrop on messages passed through the network. We assume that the goal of an eavesdropper is ultimately to discover what participants in the network are “up to”. The first type of eavesdropper is able to intercept network level communication. This eavesdropper does not participate in the network, but is able to intercept date “on the wire”. The second kind of eavesdropper actively participates in the network, and can thereby log any messages that are sent to or routed through the adversary. The first type of adversary is easily subverted using encryption; all peers in R^5N perform a public key exchange and establish a symmetric session key upon connecting to each other. Assuming this encryption cannot be broken, a network level adversary is unable to glean any useful information about peer activities.

The second type of adversary has the chance to learn more about the activities of those peers that it is connected to. Any messages routed to the adversary can be logged and possibly attributed to a specific peer in the network.[5] DHT messages are identified by a 512 bit hash code which is (generally) the one way hash of the data being inserted into the network. This hash is difficult or impossible to reverse, so simply observing a *GET* request gives the adversary little information. However, a *GET* response contains the data that was initially searched for; so the adversary could identify the relationship between a peer and data that is searched for or downloaded. R^5N does not impose restrictions on what key/data values are used, so it is possible to store data in the clear in the DHT. Our answer to this possible security issue is that any data which is sensitive should be encrypted before being inserted into the DHT, but this is a problem for higher level applications.

7.3.8.2 DoS Attacks

Our system should have protections against asymmetric denial of service (DoS) attacks where malicious peers are able to utilize more resources in bandwidth or storage than they provide. Our routing algorithm does not incorporate any kind of reputation system; nor does it use a tit-for-tat policy when routing requests. However, R^5N has two protections against malicious peers that attempt to flood the network with requests in order to use up other peers resources. First, the number of hops that a request travels is bounded by T, meaning requests do not travel around the network indefinitely. Second, malicious peers which send many *GET* requests are limited in the amount of storage which is utilized for awaiting responses at each peer. Once the pending *GET* queue is full for a particular peer, all new *GET* requests will be dropped until the queue empties based on a timeout. This limits the maximum flooded requests that any single peer can force another to handle, making it more difficult for an attacker to DoS the network by flooding inexpensive *GET* requests.

7.3.8.3 Flooding/Poisoning Attacks

As described previously, flooding the network with *GET* requests is somewhat mitigated by the per-peer queue for incoming requests. However, malicious adversaries may also flood the network with *PUT* requests in an attempt to exhaust the storage capacity of a peer or to conduct a poisoning attack. A poisoning attack [179] is a type of resource attack where an adversary attempts to replace legitimate data in the network with fraudulent or malformed data. This blocks other peers from accessing the original content when the fraudulent data is returned. R^5N allows multiple data items to be

[5] While indirection *may* prevent easy identification of message origin, a determined adversary could discover it.

inserted under the same key, so it is not easy for a malicious peer to poison a specific key. Even if many copies of fraudulent data are inserted under the same key as the original content, the correct data will still be accessible (though the receiver may have to sift through the improper responses first). The Bloom filter, described in Section 7.3.7, also provides peers the ability to filter previously seen fraudulent data. In the same manner, poisoning attacks can be prevented by utilizing self certifying data.

R^5N uses a user configured fixed size cache for storing DHT data, and thus is vulnerable to storage exhaustion attacks. While this type of attack is possible, it would generally be very expensive for an attacker and only provide a limited chance of success. Suppose a malicious peer wanted to block access to some data stored under a specific key by exhausting the storage on each peer that would be responsible for that key. The data cache that R^5N uses removes data with the soonest expiration when full when a PUT request is received for data with a longer expiration time. Also, the maximum size of any data stored in the data cache is 64 kilobytes, with a maximum expiration time of 24 hours. Thus, in order to exhaust the storage at a peer the attacker would have to transmit at least the equivalent number of 64 kilobyte PUT messages per day as the size of the user configured data cache. We can calculate the bandwidth required by attacker to fill a data cache of differing sizes as:

Remark 7.6. *Given a data cache size d (kilobytes), maximum content expiration x (hours), the attacker bandwidth required $A(d, x)$ is:*

$$A(d, x) = \frac{8 \cdot d}{3600 \cdot x} \tag{7.3}$$

Table 7.2 shows the amount of bandwidth required for various sizes of data cache. The main point is that an attacker would have to provide a relatively large amount of bandwidth for an extended period of time in order to effectively perform this type of attack. The cheapest type of memory is hard drive space, and by providing more of it to be used for the data cache this attack can be effectively mitigated. An equally effective way to combat this attack would be by decreasing the maximum allowed content expiration time, although this would require normal peers to insert data more often to keep content up to date. However, due to the high cost and low effectiveness[6] it is unlikely that an attacker would choose to perform this type of attack.

7.3.8.4 Dropping/Sybil/Eclipse Attacks

We also consider peers that attempt to disrupt normal network operations by dropping requests and/or positioning themselves in the network to block

[6] Consider that a malicious peer would have to target *all* nearest peers in a network

Cache Size	**Attacker Bandwidth 24 Hour Expiration**	**Attacker Bandwidth 12 Hour Expiration**
100 MB	9.5 Kbit/s	19.0 Kbit/s
500 MB	47.4 Kbit/s	97.1 Kbit/s
1 GB	97.1 Kbit/s	194.2 Kbit/s
2 GB	194.2 Kbit/s	388.4 Kbit/s
5 GB	485.45 Kbit/s	970.9 Kbit/s

Tab. 7.2: Table showing the trade-off between hard disk space used for the DHT data cache, maximum allowed expiration time for inserted data and attacker bandwidth required to poison the cache.

access to specific data items. Peers may drop requests for a number of reasons, including client misconfiguration, CPU or bandwidth limitations or malicious behavior. Peers which simply drop messages generally have a limited impact on overall network operation, as replication of data and requests to a number of peers means that a high proportion of peers must be dropping requests for one reason or another before users will suffer significantly.

The Sybil attack [47] is known to be one of the most detrimental attacks on DHT networks, and Kademlia in particular [170]. A Sybil attack is one in which an adversary is able to create or represent itself as multiple peers participating in the network. Simply creating many "sybil peers" is not in and of itself detrimental to network operations, so the Sybil attack is typically used as a means to better perform more sophisticated attacks. A Sybil attack may target a particular key in the DHT, for instance, where all Sybil nodes claim identifiers closer to the target than any other peers in the network. Once enough Sybils are created, the adversary can block or curtail access to the data.

Another type of attack enabled by Sybil peers is known as the Eclipse attack. In this type of attack, a specific peer or group of peers is targeted with the goal of excluding their access to the non-malicious portion of the network. This can be accomplished if the Sybil peers are able to somehow fill all of the slots in the routing table of a victim peer, excluding legitimate peers. Once this is accomplished, the victim peer is effectively partitioned from the rest of the network, and the Sybil peers can redirect requests to other malicious peers, refuse requests, etc. Some DHT routing tables are constructed in such a way that Sybil peers can force their way into routing tables by assigning identifiers in a specific range. Recent DHT designs such as Kademlia have protections against forceful routing table takeover; for instance by never replacing an older, live routing table entry with a new one. However, even this type of protection is not enough in the presence of churn coupled with a determined Sybil adversary. Due to churn, the legitimate peers in a routing table will eventually go off-line. If the adversary can keep the Sybil peers up longer than the time it takes for all legitimate peers in the victim's routing table to be churned out the routing table can still be dominated.

R^5N is susceptible to the Sybil attack insofar as peers are unrestricted in the choice of their identifiers. We also employ the same method as Kademlia, where newer peers do not replace older routing table entries once the appropriate routing bucket is full. So R^5N enjoys the same protection at minimum as that provided by Kademlia. Furthermore, in the sparsely connected restricted-route topologies that R^5N is targeted at it is much more difficult if not impossible for Sybil peers to insert themselves at random *positions* in the network. Due to the large number of "nearest" peers in such topologies, a Sybil adversary would need to create a greater number of nearest peers and place them in strategic positions (i.e. connected to certain peers) which is generally not possible in restricted-route topologies.

7.4 Mathematical Evaluation

In this section, we determine the number of hops required for finding a single nearest peer for random routing, R^5N and Kademlia. From this, we show the number of total hops that are necessary to achieve a desired success rate for *GET* requests. We demonstrate that the total number of connections in the network is the most important factor for determining the performance of these routing algorithms.

Because we are working with random graph topologies, we use the "random walk" method as a baseline routing algorithm. It has been shown [70] to be the most efficient method for searching unstructured random graphs. We compare this random routing algorithm to the Kademlia [112] routing algorithm and to R^5N.

7.4.1 Hops to Reach a Nearest Peer

The primary measure of efficiency in DHT routing algorithms is the number of hops that a request must travel in order to reach its intended destination. In our DHT, the "intended destination" of a request is any of the "nearest peers" in the network, where a nearest peer is one which fits the following definition:

Definition 7.7 (Nearest Peer)**.** *A peer i is considered nearest to random identifier l if:*

$$\bigvee_{j \in R_i} D(j,l) \leq D(i,l). \tag{7.4}$$

Our distance definition, Definition 7.1 is similar to the XOR metric of distance between two identifiers, l XOR m. However, our distance metric implies that the first non-matching high-order bit between two identifiers is the determining factor; any remaining bits are ignored. The result is that two peers with identifiers sharing the same number of matching bits with a key may *both* consider themselves nearest peers to that key. Figure 7.3 shows an example of a key in a network which has two nearest peers by our definition, but would only have one according to the "normal" Kademlia XOR metric. This definition of nearest peers provides an advantage for our replication mechanism. We rely on having enough nearest peers in the network so that we can replicate requests at multiple peers, such that two requests for the same key will overlap with high enough probability to ensure success. The increase in the number of nearest peers for multiple network topologies is seen in Table 7.1 as compared with the predicted number of nearest peers given by Definition 7.7. There is obviously a trade-off implicit in creating a greater number of nearest peers; too many will require very high replication levels, causing more traffic in the network. Too few nearest peers will make attacks by malicious peers and the behavior of benign, failing peers more detrimental to the network.

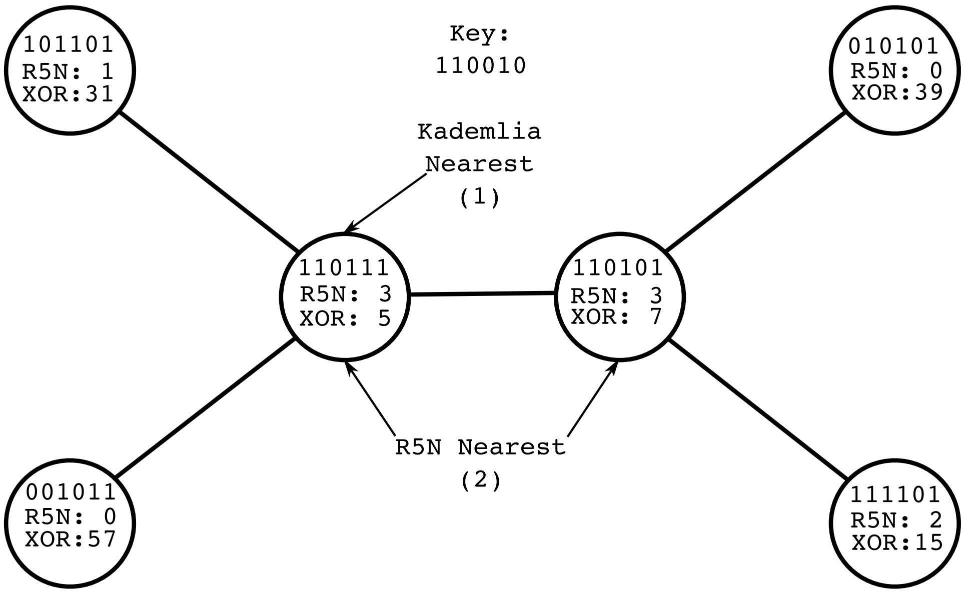


Fig. 7.3: This figure shows an example overlay network, peer identifiers and the distance between the key and identifiers for each peer. For the key shown, "110010", R^5N results in two nearest peers "110111" and "110101", whereas Kademlia's XOR metric results in only "110111" as the nearest peer. Note that the R^5N metric defines the nearest peers as those with the highest distance value, while Kademlia's XOR defines the nearest peer as the one with the least distance value.

In this section we show how the average number of hops required to reach a nearest peer can be derived based on the average number of random connections per peer in the topology (α) and the total number of peers (n). In a random graph with no overlay structure imposed, exhaustive search in the network is typically done by flooding. Flooding ensures that every peer in the network is reached by a single request (if necessary). However, random sampling via random walk routing provides better efficiency than flooding while still achieving a high level of success. Using random sampling (sending a request out a specific number of random hops in an unstructured random graph) means that requests can "fail", and in our scenario this means routing terminates before a nearest peer is found. Random sampling is repeated until the request succeeds, possibly reaching all peers in the network. Since our topologies are semi-structured random graphs, our analysis begins by determining the number of hops required for finding a nearest peer using this random walk method of routing. We then show the number of hops required to find a nearest peer for the Kademlia routing algorithm and R^5N. Finally, we move to an analysis of the *total* hops required to provide a desired probability of overlap between two requests for the same identifier, when those requests can be initiated from any two peers in the network.

7.4.1.1 Random Walk DHT Routing

The random walk routing algorithm [70] routes to a peer selected from the routing table uniformly at random at each hop. The routing table constraints we have imposed make this analysis a bit more complex than taking the probability of being at a nearest peer at any given hop as

$$\frac{1}{\alpha + 1} \tag{7.5}$$

(given α completely random connections, each peer should be equally likely to be nearest to any random identifier). However, routing tables are more likely to contain peers with bits matching the current peer. This skews the likelihood of a nearest peer being found at each step, and *reduces* the number of hops required to find a nearest peer. For this reason, we account for this difference by taking into account the possible matching bits at each of the previous hops.

The number of peers which can be used for routing is typically fewer than α due to routing table restrictions. The expected number of peers in the routing table is given by the following theorem:

Theorem 7.8 (Number of peers in routing table, $\delta_{\alpha,k}$). *Given α average random underlay connections per peer and a bucket size of k, the expected number of peers in the routing table is:*

$$\delta_{\alpha,k} := k \cdot \log_{\frac{1}{2}} \left(\frac{k}{\alpha} \right) + k \tag{7.6}$$

In order to prove Theorem 7.8, we first need to define and explain some of the primitives necessary. We start by showing the probability a certain number of bits match between any two random identifiers.

Remark 7.9 (ρ_x)**.** *Given two random identifiers l and m, the probability x consecutive, high-order bits match in the prefixes of l and m is:*

$$\rho_x := \frac{1}{2}^{(x+1)} \tag{7.7}$$

We can find the number of expected peers in any bucket z by the following:

Lemma 7.10 (Expected Peers in Bucket $\xi_{z,\alpha,k}$)**.** *Given α average random connections per peer and a bucket size of k, the expected number of peers in bucket z is: $\alpha \cdot \rho_z$ or k, whichever is smaller:*

$$\xi_{z,\alpha,k} := \min(\alpha \cdot \rho_z, k) \tag{7.8}$$

Proof: *$\xi_{z,\alpha,k}$, Expected Peers in Bucket.* Given α random connections, the probability of z bits matching between a peer and one of these random connections is given by Remark 7.9. We can therefore multiply α by ρ_z to get the expected number of peers that will go in any bucket z. However, the maximum number of peers in a bucket is k, so if the result is greater than k we limit the number of peers to k. □

To determine the probability that a certain number of bits match between a random identifier and a peer, we need to know how many buckets in the routing table are full.

Lemma 7.11 (Number of full buckets $\nu_{\alpha,k}$)**.** *Given α average random connections per peer and a bucket size of k ($\alpha \geq k > 0$), the expected number of full buckets is:*

$$\nu_{\alpha,k} := \log_{\frac{1}{2}} \left(\frac{k}{\alpha}\right) \tag{7.9}$$

Proof: *Number of full buckets $\nu_{\alpha,k}$.* We need to find the last full bucket index, which can be found by solving for the bucket index $\nu_{\alpha,k}$ in the routing table with k entries.

According to Lemma 7.10, the bucket $\nu_{\alpha,k}$ with *exactly* k entries must be the last full bucket. Any higher value of $\nu_{\alpha,k}$ will give a bucket with less than k entries, and any bucket lower than $\nu_{\alpha,k}$ will have more than k candidate entries:

$$k = \alpha \cdot \left(\frac{1}{2}\right)^{\nu_{\alpha,k}}$$

Dividing by α gives:

$$\frac{k}{\alpha} = \left(\frac{1}{2}\right)^{\nu_{\alpha,k}}, \tag{7.10}$$

Note that $\alpha > 0$.

Taking the log of both sides yields:

$$\log\left(\frac{k}{\alpha}\right) = \nu_{\alpha,k} \cdot \log\left(\frac{1}{2}\right)$$

So the expected number of full buckets is:

$$\nu_{\alpha,k} = \frac{\log\left(\frac{k}{\alpha}\right)}{\log\left(\frac{1}{2}\right)} = \log_{\frac{1}{2}}\left(\frac{k}{\alpha}\right).$$

□

We now have the required Lemmas to prove Theorem 7.8.

Proof : *Number of peers in routing table,* $\delta_{\alpha,k}$. We need to find the number of peers in a routing table, given α random connections and a bucket size of k.

Lemma 7.11 gives the number of full buckets, which we multiply by the bucket size k to get the number of peers in full buckets:

$$k \cdot \log_{\frac{1}{2}}\left(\frac{k}{\alpha}\right) \tag{7.11}$$

The remaining peers fall into non-full buckets. The first non-full bucket is assumed to have $\frac{k}{2}$ peers, the second $\frac{k}{4}$ peers, and so on. This can be expressed as:

$$\sum_{p=1}^{\infty} \frac{k}{2^p} = k \tag{7.12}$$

Adding this to the number of peers in full buckets gives:

$$\delta_{\alpha,k} = k \cdot \log_{\frac{1}{2}}\left(\frac{k}{\alpha}\right) + k \tag{7.13}$$

□

From this point on, we make the simplifying assumption that *exactly* $\nu_{\alpha,k}$ buckets are full, and that no connections exist in any higher buckets. This assumption reduces the accuracy of our formulations, as the number of peers expected to be in the routing table will be off by on average k connections. However, it allows us to predict, for instance, the exact number of matching bits expected at each hop, which would be complicated by accounting for buckets which may have between 0 and k peers. In any case, this causes us to (at worst) underestimate the number of hops which only *improves* the estimated number of hops for random routing. As our conclusion is that random walk routing is too *expensive*, this actually provides a slight advantage to random walk routing, slightly less than the bounds given by previous work [104, 132].

Now, we can calculate the probability that a peer, with a certain number of bits matching a request, is routed to.

Lemma 7.12. *The expected number of peers in a routing table that match **exactly** i bits with a random identifier, assuming j bits match between the current peer and the random identifier, given α average connections and a bucket size of k for $j < \nu_{\alpha,k}$ is:*

$$\mathfrak{e}_{\alpha,k}(i,j) := \begin{cases} (\nu_{\alpha,k} - i) \cdot k & \text{if } i = j \\ \frac{k}{2^{i-j}} & \text{if } i > j \\ k & \text{if } i < j \end{cases} \tag{7.14}$$

Proof: *We prove Lemma 7.12 by cases.* For the case where $i = j$, all buckets with index greater than i and index less than $\nu_{\alpha,k}$ will be full and guaranteed to match exactly i bits. Thus, the peers in the $\nu_{\alpha,k} - i$ buckets will match **exactly** i bits. Since there are k peers in each full bucket, we get $k \cdot (\nu_{\alpha,k} - i)$ total peers matching i bits.

For the case where $i > j$, we want to count all peers in the routing table that match more bits than the current peer. Bucket j is known to be full (because $j < \nu_{\alpha,k}$), so we expect (of those peers in bucket j), that $\frac{k}{2}$ peers will match $j + 1$ bits, $\frac{k}{4}$ will match $j + 2$ bits, and so on.

For the last case, where $i < j$, we must count all peers in the routing table that match fewer bits than the current peer, and since only one *full* bucket exists with index i, we know that exactly k peers match. □

We need to know the expected number of peers in a routing table for a peer that match a certain number of bits, b, so that we can calculate how many bits will match at the next hop, as the number of peers matching divided by the total number of peers in the routing table.

Lemma 7.13. *The probability that b bits match a random identifier at hop h in a random graph when using constrained routing tables is:*

$$P_\alpha(b,h) := \begin{cases} \rho_b & \text{if } h = 0, \\ \displaystyle\sum_{i=0}^{\nu_{\alpha,k}} \left(\frac{\mathfrak{e}_{\alpha,k}(i,b)}{\delta_{\alpha,k}} \cdot P_\alpha(i, h-1) \right) & \text{otherwise.} \end{cases} \tag{7.15}$$

Proof: *we prove Lemma 7.13 by cases.* For the base case, when at hop 0, we need to find the probability that b bits match between two random identifiers, in this case between the peer's identifier and a random key. This probability is given by Remark 7.9, as ρ_b.

For the second case, we consider each possible number of bits that may have matched at the previous hop and the associated probability. The possible number of bits that matched at the previous hop range from 0 to $\nu_{\alpha,k}$, because if more than $\nu_{\alpha,k}$ bits had matched, the previous peer would have been a nearest peer and not routed the request on. We multiply the probability of bits matching at the previous peer and the random identifier by the probability a peer existed in the previous peer's routing table with that many bits to get the probability at hop h.

For random routing, each peer is equally likely to be routed to, so the probability that a peer is present in the routing table and will be routed to is the number of peers with the proper number of matching bits ($\mathfrak{c}_{\alpha,k}(i,b)$) divided by the total number of peers in the routing table ($\delta_{\alpha,k}$). $\square$

Lemma 7.14. *The probability of routing for h hops without previously encountering a nearest peer and the peer at hop h being a nearest peer, given α connections per peer is:*

$$\grave{\varphi}_\alpha(h) := \begin{cases} 1 - \sum_{b=0}^{\nu_{\alpha,k}} P_{b,h} & \text{if } h = 0, \\ \left(1 - \sum_{b=0}^{\nu_{\alpha,k}} P_{b,h}\right) \cdot (1 - \sum_{i=0}^{h-1} \grave{\varphi}_\alpha(i)) & \text{if } h > 0. \end{cases} \tag{7.16}$$

Proof : *we prove $\grave{\varphi}_\alpha(h)$ by cases.* The base case is the probability of being at a nearest peer at the first hop ($h = 0$). We calculate this probability as the sum of the probabilities that b bits match between a random identifier and the current peer for values of $b = 0$ to $b = \nu_{\alpha,k}$. The probability for each of these values of b, $P_{b,h}$, is the probability that a peer exists in the routing table and will be routed to. This is the probability that a peer is *not* a nearest peer. We subtract this value from 1 to get the probability that the peer at hop $h = 0$ *is* a nearest peer.

We do the same summing of probabilities at subsequent hops $h > 0$, only we must multiply the result by the remaining probability that hop h was ever reached. This remaining probability can be found by taking the total starting probability, 1, and subtracting the sum of the probability of each peer at each previous hop $\grave{\varphi}_\alpha$ being a nearest peer. $\square$

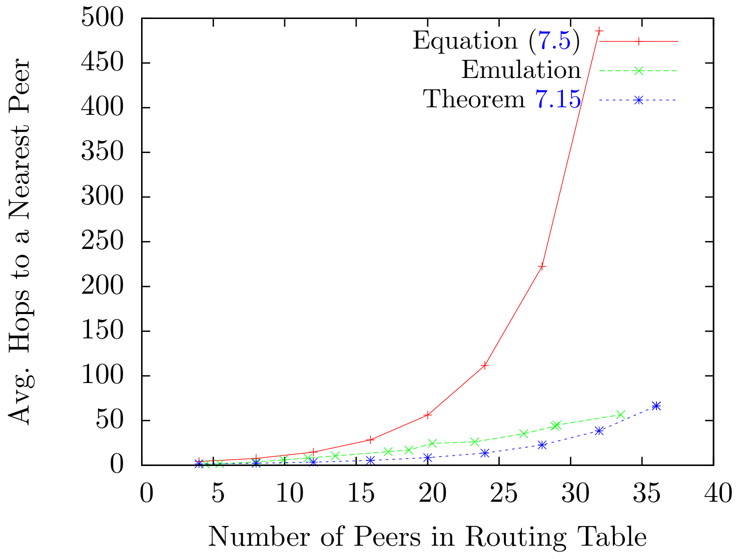
Theorem 7.15 ($\grave{\imath}_\alpha$)**.** *The expected number of hops to reach a nearest peer* *using random routing given α connections and constrained routing tables is:*

$$\grave{\imath}_\alpha := \sum_{h=0}^{\infty} h \cdot \grave{\varphi}_\alpha(h) \tag{7.17}$$

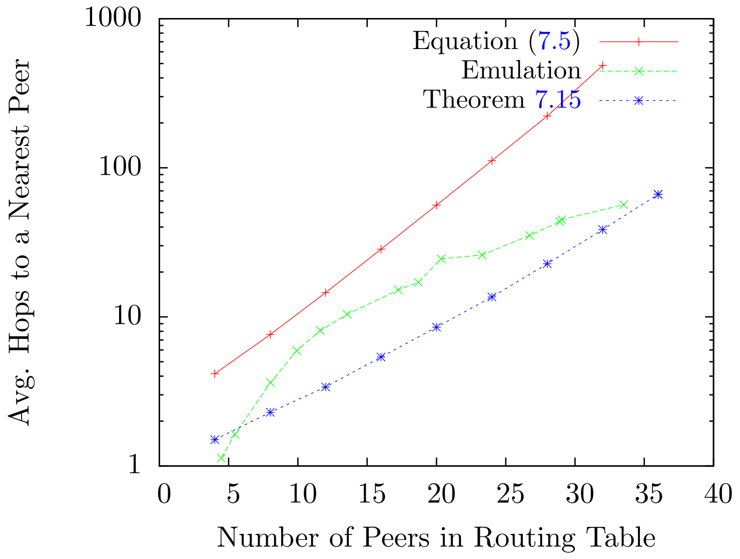
Proof : $\grave{\imath}_\alpha$. Taking the sum of multiplying the hop count by the probability of reaching a nearest peer at every hop (from zero to infinity) gives us the *expected* number of hops required to reach a single nearest peer. □

Theorem 7.15 allows us to calculate the number of hops required to find a single nearest peer using random routing with routing tables constrained as described in Definition 7.2. It is important to note that Theorem 7.15 gives a bound for random routing which is linear in relation to α (which is also a measure of the total number of connections in the network), which is proven in previous work [104,132] and supported by our findings in Figure 7.4. The implication of this is that the network *size*, n, has no impact on the number of hops that are required to reach a nearest peer. If the underlay topology is unrestricted, such that $\alpha = n$, then random routing is $O(n)$, provided that each peer can only be encountered once on the random walk. Most research on random routing [104, 110, 132] therefore gives a bound of *greater* than $O(n)$ in the worst case, assuming that there is only a single nearest peer (or "replica") in the network. Clearly in this case it is possible that all peers will be traversed (some multiple times) before reaching the single nearest peer.

The mathematical model we have thus derived for random walk routing in networks with Kademlia routing tables is complex, but we believe that it is necessary. A simpler method for calculating the number of hops is based on Equation (7.5). In fact, this was the initial model we used, but found that it did not meet up with results from routing in practice. This necessitated the more complex Theorem 7.15, which takes into account routing table constraints on α possible connections. Figures 7.4a and 7.4b plot the expected number of hops for Equation 7.5, Theorem 7.15, and the average number of hops from an emulated test network with $\alpha = n$ connections (an unrestricted clique underlay topology resulting in routing tables size $O(\log n)$). It is clear from this data that using Equation (7.5) for random routing overestimates the number of hops required for finding nearest peers, as explained by our routing table restrictions. Figure 7.4 plots the number of usable routing table entries per peer on the x-axis (approximately $\log n$), and the average hops required to reach a nearest peer on the y-axis. When plotted without scaling in Figure 7.4a, the lines seem to grow exponentially with the peers in the routing table; that is because the network size grows exponentially as the number of routing table entries increase.



(a) Default plot scale



(b) Log scale

Fig. 7.4: Comparison of random routing from Equation (7.5) (clique topology with $n = \alpha^2$ peers), constrained random routing (Theorem 7.15) and the observed results from emulation. Our model (Theorem 7.15) is more accurate than Equation (7.5).

In conclusion, imposing routing table constraints on random walk routing reduces the average number of hops required to find a nearest peer. The average number of hops required to find a nearest peer grows linearly with the number of nearest peers in the network. The number of nearest peers in the network is a direct result of the average number of random connections (α). This is similar to the result found in previous analyses of random walks on graphs [104]. Next, we analyze the Kademlia routing algorithm to show expected number of hops for Kademlia routing.

7.4.1.2 Kademlia Routing

We now model the Kademlia routing algorithm to find the expected number of hops to find a single nearest peer. Kademlia differs from random routing in that at any given hop, either a nearest peer will be found, or the request will be forwarded to a routing table entry with one or more additional matching bits. Having a peer with a least one additional matching bit is guaranteed by Definition 7.7 provided the current peer is not a nearest peer. This allows us to deduce that at hop h, at least h bits (possibly more) are known to match between the current peer's identifier i, and the identifier being searched for, l. This reduces the set of possible decisions that can be made at hop h (only peers in higher buckets will be considered), which makes it more likely (compared to random search) at any given next hop that a nearest peer will be found, thus reducing the number of hops required overall.

This analysis is nothing novel; in fact, the original authors of the Kademlia routing algorithm proved equivalent tight bounds [112] on the performance of the Kademlia greedy routing algorithm as those we provide. However, their bounds were based on an unrestricted clique as the network underlay topology. We provide the analysis in this section to show that we can bound routing based only on α and Definition 7.7 instead of XOR as the distance metric. For the case where $\alpha = n$, our results are the same as theirs.

Remark 7.16 ($\varepsilon(z, \alpha)$)**.** *Given α total random connections at a peer, bucket z will be empty with probability:*

$$\varepsilon(z, \alpha) := (1 - \rho_z)^\alpha \tag{7.18}$$

We perform an analysis of Kademlia routing in the average case. For this analysis, we want to find the average number of hops we expect before finding a nearest peer. We know at each hop that at least h bits match between the current peer and the search key. However, we must account for the possibility that one *or more* bits match between the random identifier and the peer chosen for the next hop from the routing table. We again make the simplifying assumption that the expected number of buckets are full and beyond that all buckets are empty.

Lemma 7.17 (Expected matching bits)**.** *Given a peer i, random identifier l and $|R_i(D(i,l))| = k$, the expected greatest number of additional matching bits between l and some $j \in R_i(D(i,l))$ is $\log_2 k$.*

Proof of Lemma 7.17. The z term in ρ_z gives the expected number of bits which match between two random identifiers. Since we know that there are k entries in the bucket corresponding to the number of matching bits at the current peer, we multiply ρ_z by k to account for each possible entry. By setting $k \cdot \rho_z$ equal to 1 (the total probability) and solving for z in ρ_z, the result is the number of bits that we *expect* will match between some peer in the bucket. Hence:

$$k \cdot \frac{1}{2}^z = 1$$

We divide both sides by k and take the log giving:

$$\frac{1}{2}^z = \frac{1}{k},$$

Note that $k > 0$.

$$\begin{aligned} \Leftrightarrow z \cdot \log \frac{1}{2} &= \log \frac{1}{k} \\ \Leftrightarrow z &= \frac{\log \frac{1}{k}}{\log \frac{1}{2}} \\ \Leftrightarrow z &= \frac{-\log k}{-\log 2} = \log_2 k. \end{aligned}$$

□

We can thus assume that on average $\log_2 k$ bits match at each hop up to the point when a nearest peer is found.

Remark 7.18. *Therefore, at hop h the expected number of matching bits is $h \cdot \log_2 k$ (because $\log_2 k$ bits matched at each hop from 0 to h), up to the point where the corresponding bucket for the number of matching bits is empty, when routing terminates.*

Now that we know the number of bits that should match at each hop, we can combine that with our routing termination condition (where the bucket which should be used for routing is empty) to find the number of hops we expect requests to take in order to find a nearest peer utilizing the Kademlia routing algorithm.

Lemma 7.19 (Expected number of hops)**.** *Given α average random connections per peer and a bucket size of k, the expected number of hops required to find a nearest peer using Kademlia routing is:*

$$\hat{\iota}_\alpha = \frac{\log_{\frac{1}{2}}\left(\frac{k}{\alpha}\right)}{\log_2 k} \tag{7.19}$$

Proof: *Expected number of hops.*

A nearest peer is found when

$$R_i(d) = \emptyset$$

By Theorem 7.8 when:

$$d \geq \log_{\frac{1}{2}}\left(\frac{k}{\alpha}\right) : R_i(d) = \emptyset.$$

So a nearest peer is found when:

$$h \cdot \log_2 k \geq \log_{\frac{1}{2}}\left(\frac{k}{\alpha}\right).$$

Solving for h:

$$h \cdot \log_2 k \geq \log_{\frac{1}{2}}\left(\frac{k}{\alpha}\right)$$
$$\Leftrightarrow h \geq \frac{\log_{\frac{1}{2}}\left(\frac{k}{\alpha}\right)}{\log_2 k}.$$

So:

$$\hat{\iota}_\alpha = \frac{\log_{\frac{1}{2}}\left(\frac{k}{\alpha}\right)}{\log_2 k}$$

□

This section provided an analysis of the Kademlia routing algorithm given our "nearest peer" termination condition. We have shown that routing with Kademlia finds nearest peers in a number of hops logarithmic with respect to α. This is the same bound as provided by the original Kademlia analysis [112], except we do not assume that $\alpha = n$, where n is the number of total peers in the network. Routing using the Kademlia algorithm therefore grows logarithmically with the number of possible *connections*. This is clearly more efficient in terms of the number of hops required than random routing; although we explain in Section 7.4.3 why this is not necessarily better for restricted-route networks. Before this explanation, we introduce the analysis of R^5N, so that we have three common points of comparison.

7.4.1.3 R^5N Routing

The routing function of R^5N performs random routing for a certain number of hops and then switches to deterministic (Kademlia) routing after the set number of hops. The number of random hops is controlled by the convergence modifier c (which should be equal to $T = \log n$). The Kademlia routing phase should then be quite efficient at finding a nearest peer. These two phases combine to give us a wider range of nearest peers in the network, with fewer hops than completely random routing.

In the following analysis, we assume that if the random routing phase of R^5N does not reach a nearest peer, then the starting peer (at hop c) for Kademlia routing is considered to be a uniformly selected random peer in the network. We implicitly made this assumption with our Kademlia analysis, as at the start of routing each peer identifier and search key are assumed to be random. As we described in Section 7.4.1.1, due to our routing table construction technique, there is some bias even in the random routing phase. However, we do reach a peer that is as random as possible, and making the assumption that we are at a random peer has little effect overall. We show this empirically in Section 7.6. Finally, in this section we find the number of hops to reach a nearest peer in the network; any bias in the random routing phase will actually *decrease* the total number of hops required, so at worst we overestimate the hops required.

Lemma 7.20. *For R^5N, the probability that a nearest peer is found at hop h, given convergence modifier c is:*

$$\bar{\varphi}_\alpha(h) := \begin{cases} \varphi_\alpha(h) & \textit{if } h < c \\ \hat{\varphi}_\alpha(h - c) & \textit{otherwise} \end{cases} \tag{7.20}$$

Proof : *Lemma 7.20 can be proven as follows:* Lemma 7.20 is the result of a straightforward combination of the probability that a nearest peer is found using the random walk routing algorithm, $\varphi_\alpha(h)$, for the first c hops; then we use the probability of finding a nearest peer from the Kademlia routing algorithm, $\hat{\varphi}_\alpha(h)$, for the second phase of routing for any hops where $h \geq c$. □

We formulate the expected number of hops for R^5N routing by taking the expected hops up to c, using the random routing formula from Theorem 7.15, and then adding the expected number of hops from Kademlia style routing, multiplied by the remaining probability that a nearest peer is not found after c hops using randomized routing. This is expressed in the following Theorem.

Theorem 7.21. *For R^5N routing, the expected number of hops to find a nearest peer is:*

$$\bar{\iota}_\alpha := \sum_{h=0}^{c} [h \cdot \varphi_\alpha(h)] + (1 - \sum_{i=0}^{c} \grave{\varphi}_\alpha(i)) \cdot \hat{\iota}_\alpha \tag{7.21}$$

Proof : *Lemma 7.21:* The first summation calculates the expected hops that will be traversed using random routing in the first c steps.

The second summation, explained in the proof of Lemma 7.14, gives the probability that one of the previous hops from $h = 0$ to $h = c$ was a nearest peer. We subtract this value from 1 to get the probability that *none* of the peers encountered during the randomized routing phase were nearest peers. We then multiply this probability by the expected number of hops for the Kademlia routing phase, $\hat{\iota}_\alpha$, to get the expected number of hops which will be routed in the Kademlia phase after the random phase.

These two separate expected hop values are then summed to give the total expected hops for the *combined* routing of R^5N. □

The main function of the R^5N routing algorithm that determines efficiency is the Kademlia routing algorithm. As described in Section 7.4.1.2, the hops required using Kademlia increase logarithmically with α. Because we add on a constant, c, finding a nearest peer using R^5N takes $c + O(\log \alpha)$ steps, regardless of the network size n (though clearly α can be no greater than n). c is a small constant, logarithmic to the size of the network; typically $O(\log n)$ for our target network topologies. So the number of hops required for R^5N is $O(\log n) + O(\log \alpha)$. Because n must be greater than or equal to α, the average number of hops to find a single nearest peer is $O(\log n) + O(\log n) = O(\log n)$.

The previous three sections have provided an overview of the performance of random, Kademlia and R^5N routing in terms of the number of hops required to find a single nearest peer. However, for any network other than a clique, there will be more than one nearest peer in the network. We may need to find more than a single nearest peer for each request; specifically we need to encounter enough peers on *PUT* requests so that a subsequent *GET* requests have a reasonable probability of success. The next section provides details on the number of peers required to ensure success, and compares the three routing algorithms based on the *total* number of hops required for each request.

7.4.2 Total Hops – Routing with Sufficient Replication

The number of hops required to find a single nearest peer is the typical metric for determining efficiency in unrestricted structured overlay networks. This is because in such networks, there will only be a single nearest peer in the network, and storage and lookup can be correctly directed to this single node for every search. However, as we have shown, restricted-route semi-structured networks generally have more than one nearest peer for any given key; all of which are the "correct" peers to handle requests for that key.

In such restricted underlay network topologies, ensuring success requires replicating requests at multiple peers. One such replication strategy would be to require PUT requests to replicate data at *all* nearest peers for a given key, such that a GET request directed to any of these nearest peers would succeed. The converse would also work; replicating PUT requests to only a single nearest peer and GET requests at every nearest peer.

However, our replication strategy only needs to provide probabilistic success guarantees because both PUT requests and GET requests are meant to be repeated (PUT requests for replication/refresh and GET requests for increasing success rates). Repeating PUT requests and periodically refreshing data is also necessary to combat natural churn in the network, so that data is not lost when the peers storing it go off-line. Thus, we can be even less stringent in our replication strategy. We can reduce the number of total hops per request drastically by only requiring roughly a 50 percent success rate for a GET request. By repeating GET requests, **randomized** search can quickly increase the success rate (the probability of *not* finding a replica decreases exponentially). According to the birthday paradox, we can reach this desired success rate by replicating PUT requests at approximately $\sqrt{\frac{n}{\alpha+1}}$ nearest peers. Now that we have a target replication factor to use for requests, we can move on to examining the total hops required for each of the three routing algorithms already described.

We make the simplifying assumption that all three routing algorithms are able to find *distinct* nearest peers up to the desired replication level. As we discuss in the next section, this is certainly not always the case; especially for Kademlia. However, it simplifies the comparison as far as the total number of hops that would required for each routing algorithm. Thus, we assume for all three routing algorithms that we need to replicate PUT requests and GET requests at the same number of peers. Calculating the total number of hops needed is a straightforward multiplication of the required replicas with the following.

Remark 7.22. *The probability of two independent requests for the same key overlapping in at least one peer when each request encounters $\sqrt{\frac{n}{\alpha+1}}$ distinct nearest peers is:*

$$1 - \left(\prod_{r=0}^{\sqrt{\frac{n}{\alpha+1}}} \frac{\frac{n}{\alpha+1} - \sqrt{\frac{n}{\alpha+1}} - r}{\frac{n}{\alpha+1}} \right) \tag{7.22}$$

The birthday paradox provides the following remark:

Remark 7.23 ($\hat{\varsigma}_{n,\alpha}$ Number of replicas)**.** *Given $\frac{n}{\alpha+1}$ nearest peers in the network, in order to ensure that two requests for the same key overlap on average 50 percent of the time, the number of replica nearest peers required is:*

$$\hat{\varsigma}_{n,\alpha} \geq \sqrt{\frac{n}{\alpha+1}} \tag{7.23}$$

7.4.2.1 Random Walk Routing — Total Hops

Remark 7.24 (Total Hops for Random Walk DHT Routing with Replication)**.** *The expected total number of hops required to find $\hat{\varsigma}_{n,\alpha}$ nearest peers using random routing in a network of n peers with α random connections is:*

$$\tilde{\iota}_{\alpha} \cdot \hat{\varsigma}_{n,\alpha} \tag{7.24}$$

The number of hops required to find a nearest peer using random routing increases linearly as the number of nearest peers in the network decreases. The network size n and the average number of connections α determine the number of nearest peers in the network. Thus, the worst case complexity is $O(\frac{n}{\alpha})$ for a single request to reach a nearest peer, and is therefore $O(\frac{n}{\alpha} \cdot \sqrt{\frac{n}{\alpha}})$ in order to provide the desired level of replication.

7.4.2.2 Kademlia Routing — Total Hops

Remark 7.25 (Total Hops for Kademlia Routing with Replication)**.** *The expected total number of hops required to find $\hat{\varsigma}_{n,\alpha}$ nearest peers using Kademlia routing in a network of n peers with α random connections is:*

$$\hat{\iota}_{\alpha} \cdot \hat{\varsigma}_{n,\alpha} \tag{7.25}$$

Our evaluation of Kademlia (Section 7.4.1.2) when used in restricted-route topologies led us to the determination that Kademlia is bound by α, routing in $O(\log \alpha)$ steps. Thus, the bound for routing to our desired number of replicas is $O(\log \alpha \cdot \sqrt{\frac{n}{\alpha}})$ for Kademlia.

However, using R-Kademlia, routing is unlikely to be able to reach the desired number of replicas for many topologies. The first problem with Kademlia is that each request for a key initiated from the same peer will always take the same route. As such, whichever replicas are found for the first request will be found with each subsequent request. Second, in sparse topologies, two requests initiated from two distant (with regard to underlay hops) peers may never overlap in *any* peers. Re-issuing requests can never solve this problem, and increasing the replication level will only work up to a point.

For Kademlia routing, replication is entirely controlled by the initial branching factor (the number of parallel requests sent out at hop $h = 0$). This allows Kademlia routing to reach more than a single nearest peer. However, each peer i has only an expected $|R_i| \leq \alpha$ total connections, and the initial branching factor cannot be greater than $|R_i|$. Thus, in the average case, the initial branching factor cannot be greater than α (and will usually be less due to routing table construction). It is also possible that parallel routes may converge to the same paths, and thus to the same nearest peer, further reducing the number of distinct nearest peers found. Clearly, Kademlia routing cannot find more than $|R_i|$ nearest peers for any given request using Kademlia greedy routing. The result of this is that if the number of nearest peers necessary to ensure requests will overlap is greater than $|R_i|$, Kademlia will not be able to ensure success. We elaborate on this problem, and show empirical results which demonstrate the problem concretely in Section 7.6.

7.4.2.3 R^5N Routing — Total Hops

Remark 7.26 (Total Hops for R^5N Routing with Replication)**.** *The expected total number of hops required to find $\hat{\varsigma}_{n,\alpha}$ nearest peers using R^5N routing in a network of n peers with α random initial connections is:*

$$\bar{\iota}_\alpha \cdot \hat{\varsigma}_{n,\alpha} \tag{7.26}$$

As we found with Kademlia, R^5N is also bounded by the average number of connections per peer, with a bound of $O(\log \alpha)$, in the case of finding a single nearest peer. This gives R^5N the same bound on total hops per request including replication as Kademlia, $O(\log \alpha \cdot \sqrt{\frac{n}{\alpha}})$.

R^5N is clearly less costly in terms of finding a single nearest peer and multiple nearest peers than the random walk method of DHT routing. While both random routing and R^5N are able to reach enough nearest peers in the network to satisfy replication requirements, R^5N's efficiency makes it the likely choice for restricted-route networks with structured routing tables.

While asymptotically equivalent, R^5N operates better in sparsely connected restricted-route topologies that Kademlia. This is because R^5N does not suffer from the replication problems inherent in Kademlia. First, R^5N is able to route to different nearest peers on subsequent requests, due to the randomized phase of routing. Therefore, requests that fail in R^5N can be repeated and will eventually find enough nearest peers to reach the desired replication level. This solves Kademlia's problem of repeat requests always encountering the same nearest peer.

Furthermore, R^5N's replication strategy relies on requests branching (see Section 7.3.6) at multiple peers along the request route, not just at the initiating peer. This allows requests to reach more distinct nearest peers than the total number of connections in the initiating peer's routing table.

This allows peers with few connections to reach enough distinct nearest peers in the network to satisfy replication requirements.

7.4.3 Comparison and Discussion

We have defined the number of replicas which are required to ensure success of a *GET* request after a single previous *PUT* request in Lemma 7.23 in, on average, 50 percent of cases. Random routing is more expensive than R^5N and Kademlia is unlikely to find enough distinct nearest peers in the network to provide the appropriate level of replication. An important variable in R^5N routing is c, which we have previously mentioned should be equal to $T = \log n$. The next section details the reasoning behind this choice for c.

7.5 Markov Mixing Times

Our assumptions about the performance of R^5N are based on being equally likely to reach any of the nearest peers in the network to a given key. In order to ensure that this holds (or that each nearest peer is as likely as possible, given a specific network topology) we need to route for enough hops in the random phase of routing so that the deterministic search “begins” at a random peer. For this we need to make c large enough that we find a random-as-possible peer in the network. In order to justify the c chosen we turn to Markov mixing theory.

A Markov chain or Markov process is any set of transitions on some state space where each transition depends only the current state. Typically a Markov chain is denoted by $\mathfrak{M} = (\Omega, \Psi(i,j))$ where Ω represents a state space and $\Psi(i,j)$ is the probability to move from i to j.

In terms of random routing on a graph $G = (V,E)$ (where V is the set of vertices and E the set of edges in G), the simple random walk is defined as follows [4]:

Remark 7.27 (For any node $i \in V$ with d_i as the degree of i)**.**

$$\Psi(i,j) = \begin{cases} \frac{1}{d_i} & \textit{for } (i,j) \in E, \\ 0 & \textit{otherwise.} \end{cases} \tag{7.27}$$

It has been shown [104] that such a simple random walk for any connected graph G has a stationary distribution, π, such that $\pi\Psi = \pi$. Once the stationary distribution is reached each state has the same probability of being reached, regardless of how many more steps are taken. The number of steps that it takes to reach such a stationary distribution is known as the *mixing time* of the network. In terms of the simple random walk as described above this is the number of hops a random walk must take in order to find any peer with unchanging probability. For any graph G and any starting peer, once this number of hops is taken, a “random-as-possible” peer has been found.

There has been much previous work on the mixing times [4,5,38,100,104, 106,185] of various graphs; so we base our analysis on those previous results. Since we particularly want our routing algorithm to be comparable to other DHT routing algorithms in highly structured networks, we concentrate our analysis on Small-World and Erdős-Rényi random graphs. The observed mixing time for any so called “fast-mixing graphs” [38] is $O(\log n)$ where n is the size of the network. The Small-World topologies that we use in our evaluation are known to be such fast-mixing graphs. The mixing time of random Erdős-Rényi graphs has been shown to be between $O(\log n)$ and $O(\log^2 n)$ [15, 67], depending on average node degree.

These mixing times, from empirical studies and mathematical evaluation, are the basis for the c parameter we use in Section 7.6. As described in Section 7.3.4, we plan to eventually use a heuristic network size estimate to bound c in practice. However, in our testing we know the network size, and can therefore precisely specify c, making our results more stable than relying on the network size estimation. The focus of this thesis is on the performance of R^5N, and the network size estimation is tangential to that evaluation. We use a value of c set to $\log n$ for our evaluation. Note that this is the *smallest* mixing time for any of our graphs, so in a sense this provides a worst case scenario for R^5N. If the mixing time is actually higher than c, requests will not be sufficiently randomized before becoming deterministic and performance will suffer.

7.6 Experimental Results

In this section, the presented experiments were generally done using a network of 2025 peers with a fixed replication level of $r = 10$ and a fixed network size estimate parameter $T = 4$ ensuring that only the shape of the topology and the node degree are parameters for the evaluation. There are two reasons for the choice of a network size of 2025. First, approximately 2000 peers are the largest number we could run on a single workstation[7] concurrently being used for other development tasks without significant performance degradation. Second, one of our Small-World topologies is based on a 2d-torus which is best constructed as an m x m grid. In this case, we used $m = 45$, yielding a total graph of size 2025. Results from large scale tests are presented in more detail in Section 7.7. As shown in Figure 7.5, R-Kademlia peaks in our base topologies with a total number of replicas somewhere around 10 (regardless of r). It would be unfair (to R-Kademlia) to use a replication target much higher than 10 as it would bias the results in favor of R^5N. We showcase the ability of R^5N to achieve higher replication separately from the trials

[7] Albeit a quad core Xeon with 12Gb of RAM.

which concern overall performance. The T parameter of 4 was chosen based on experimental results using the R-Kademlia algorithm. When we allowed an unrestricted topology and peers exchanged *FIND PEER* messages until routing tables converged; the average number of hops for *PUT* and *GET* requests was a bit under 4. Therefore, we expect such a topology to require around 4 hops in ideal conditions. Using this ideal value of $T = 4$ again gives the advantage to R-Kademlia; R^5N may require more than 4 random hops in order to reach a truly random peer in the network depending on the topology. Failing to reach a random peer in the initial phase of routing is detrimental to R^5N as it reduces the number of nearest replicas that can be reached over time. However, failing absolute knowledge of the topology (which we do not assume peers have), each peer must make an estimate. In order to reduce CPU utilization (and thus further limit scalability), we did not use the distance vector transport from Section 7.3.2 to improve connectivity; instead, we report results for a range of node degrees in the underlay topology. These topologies encompass those achieved using the distance vector transport on less well-connected topologies.

In this section, we will first report on some relevant specifics of our implementation and the simulation environment, then discuss the R-Kademlia implementation that we use as a point of reference to the original Kademlia routing algorithm and finally present the result data.

7.6.1 Implementation Details

We have implemented our DHT routing algorithm on top of GNUnet, GNU's framework for secure P2P networking. GNUnet is extended by developing new *service* processes that are responsible for particular functions, such as handling P2P messages of a particular type. A typical GNUnet peer consists of about a dozen different services that coordinate using IPC.

Due to the use of different processes, GNUnet services are largely isolated against faulty behavior of other parts of the framework. The R^5N DHT service primarily uses link-encrypted communication between nodes as a foundation. The P2P framework uses transport plugins to enable low-level peer-to-peer communications. The framework typically communicates over the Internet (e.g., TCP and UDP). The onion-routed fisheye distance vector layer is implemented as another transport plugin. GNUnet uses 512-bit hash codes and hence peer identities and keys for our implementation are 512-bits.

In order to improve performance in practice, our implementation diverges slightly from the idealized description of the replication mechanism from Section 7.3.6. We continue to forward to $\Upsilon_{r,h}$ peers for $h \geq 2 \cdot T$ instead of just until $h < T$. On the other hand, *PUT* requests are only forwarded until a nearest peer is found and *GET* requests are not forwarded if a definitive result has been found. This heuristic accounts for path collisions and inaccuracies in the network size prediction. In our experiments, it still meets the replication target r by a margin of error of a factor of 2 for large networks.

As alluded to in Section 7.3.3, there are two types of generic routing messages used by our implementation; outgoing requests and request results which are routed back to the initiator. The actual DHT request type (currently *GET*, *PUT* and *FIND PEER*) is opaque inside of the generic outgoing request and result messages; as a result, new request types can be forwarded by peers even if they do not support the underlying request type.

Outgoing requests contain various message options, including the desired replication level which is capped by a system-wide limit to prevent abuse. Other message options specify how each peer that receives the request should process it; for instance, *FIND PEER* messages can optionally be handled by each peer along the path as opposed to the default behavior of being handled only at locally nearest peers. This can help build routing tables faster for a network that is under little load. Of course, there is no way to enforce that intermediate peers do the proper handling (nor would we want them to). For instance, if a *PUT* request is sent with the flag indicating the data should be stored at every peer that receives the request; it makes sense for peers to decide locally whether or not they choose to store the data.

7.6.2 Emulation Framework for Testing and Profiling

We have analyzed the expected performance of R^5N using mathematical analysis, simulation and emulation; but the experimental results presented in this section use emulation. These are attained using the newly developed testing and profiling framework described in detail in Chapter 6. Using this framework, we can emulate tens of thousands of peers running the actual, unmodified implementation. Many peers are emulated on a single host and many such hosts can be combined to emulate larger networks. Performing large scale experiments is quite important for verifying that the routing algorithm performs as desired with realistic numbers of peers, and for proving that our emulation framework operates as designed. However, the large scale and small scale results largely agree; thus we concisely demonstrate our overall results here, and give an overview of many more tests in Section 7.7. The testing framework can generate various network underlay topologies, emulate churn and evaluate the performance of the DHT and other applications. Complete details of the testing and emulation framework can be found in Chapter 6.

7.6.3 R-Kademlia

We use a variant of Kademlia, which we call R-Kademlia[8], as a point of reference. The iterative routing in the original Kademlia design performs so badly in a restricted-route topology that it is not useful for comparison. While peers cannot connect arbitrarily, it would be possible to use special RPC messages to instruct distant peers about routing choices. However, doing so would increase message cost drastically and there is no way to guarantee distant peers behave as instructed. Therefore we have implemented R-Kademlia, a recursive implementation of Kademlia that is otherwise as faithful to the original design as possible.

The biggest problem with a recursive implementation of Kademlia is that r concurrent requests are meant to be kept in flight until no closer peers are found. R-Kademlia initiates r messages at the first peer, but can only promise that typically r nearest peers will be found, as requests terminate once a nearest peer is reached. If the topology is a clique, these r peers will be the globally closest to the key. Of course, as described in Section 7.3.6, finding a diverse set of nearest peers can be beneficial in sparse topologies and R-Kademlia benefits from this as well. Peers in R-Kademlia are responsible for attempting to forward requests only to peers that have not encountered the request already. This is facilitated by a Bloom filter, included with each request, which stores the peers previously encountered on the route (as explained in 7.3.3). Peers also maintain a limited store of recent requests; thus, if the same request reaches a peer twice, the Bloom filters are merged. Using these techniques, we mimic the iterative routing of Kademlia, with the exception that the initiator cannot control next-hop decisions.

A small difference between Kademlia and R-Kademlia is in the way *FIND PEER* requests are sent and handled. In Kademlia every peer that a *FIND PEER* request reaches sends its full routing table back to the initiator. With R-Kademlia, doing so is impractical because it might require a large amount of (possibly redundant) information to be sent back down long query paths to the origin. Instead, only locally closest peers send their full routing tables; other peers on the path attempt a connection to the sender of the request if the respective bucket in their own routing table is not full.

Also, peers are responsible for deciding when to stop sending *FIND PEER* messages (based on diminishing returns). As another optimization, *FIND PEER* messages include a Bloom filter containing peer identities that the initiator is already connected to. This also reduces the amount of routing information sent from even the closest peer found. The reason we do not include contact information in all route requests is a matter of efficiency. The use of multiple transports in GNUnet means that a single peer may have many different addresses; including this information (which connected peers

[8] Not to be confused with [79], a recursive Kademlia implementation for simulation

already know) would be redundant and impose an unnecessary burden on peers.

7.6.4 Network Performance

For networks with few connections, the success rate of R^5N is significantly higher than it is for R-Kademlia. The worst case for R^5N when compared to R-Kademlia is hence an unrestricted underlay topology (clique). In this case, both R-Kademlia and R^5N will always find the data at the nearest peer on the first attempt, but R^5N is expected to take longer. Table 7.3 shows the average number of hops taken for the two algorithms in this worst-case scenario for R^5N.

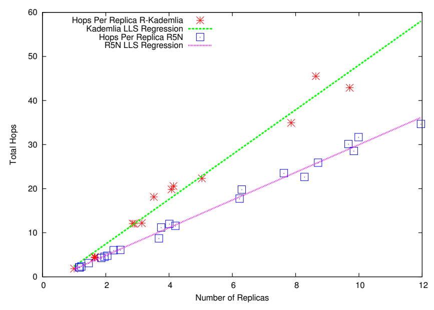
Size of network	Average hops per *PUT*		Average hops per *GET*	
	R-Kademlia	R^5N	R-Kademlia	R^5N
100	2.70 ± 0.06	3.96 ± 0.06	2.54 ± 0.03	4.63 ± 0.17
250	3.06 ± 0.10	4.26 ± 0.10	3.10 ± 0.06	5.96 ± 0.27
500	3.08 ± 0.46	4.38 ± 0.45	3.38 ± 0.06	6.17 ± 1.14
750	3.19 ± 0.74	4.37 ± 0.83	3.50 ± 0.04	6.29 ± 1.04
1000	3.63 ± 0.07	4.47 ± 0.93	3.64 ± 0.04	7.29 ± 0.95

Tab. 7.3: Average hops for R-Kademlia and R^5N in clique underlay topologies of different sizes. As expected, R^5N takes about twice as many hops as R-Kademlia.

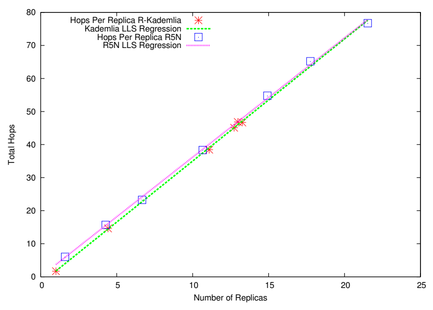
7.6.5 Replication

As described in Section 7.6.3, R-Kademlia attempts to achieve a certain replication level r by sending r parallel requests from the initiating peer. In contrast, R^5N probabilistically chooses multiple peers to forward the request to at each hop. Neither approach is able to precisely hit the specified replication target; however, R^5N produces the same number of replicas with significantly fewer messages when compared to R-Kademlia (Figure 7.5) for unrestricted and Small-World topologies, and about the same number of total messages for the other two topologies. This is because sending out many parallel requests from the same initial peer increases the chance that paths will at times converge, while requests that branch at later hops are likely to be further apart in the network and carry more information about which peers have already been routed to and therefore overlap with lower probability. We limit results to 30 total replicas or less, at higher replication levels R^5N outperforms R-Kademlia in all topologies[9].

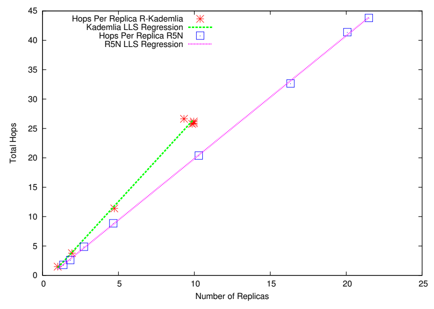
[9] Due to R-Kademlia's inability to create more replicas than connections.



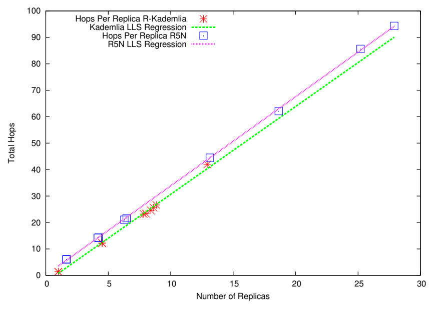
(a) Unrestricted



(b) 75% NATed



(c) Small-World

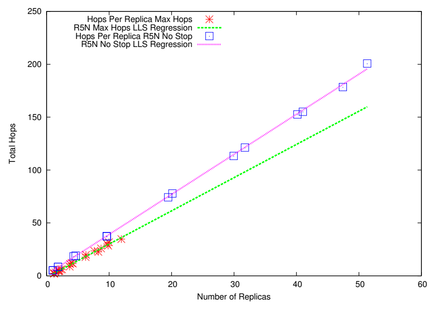


(d) Random Graph

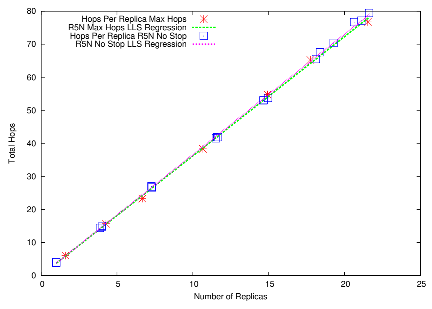
Fig. 7.5: Average hops required per replica; varying replication level r.

Figure 7.6 compares the total hops required per replica when PUT requests do not stop based on the condition where $h > 2 \cdot T$. When viewed in contrast to Figure 7.5, this motivates our implementation optimization of ending PUT requests when a nearest peer is encountered. In this figure, we see R^5N is more costly than R-Kademlia for the unrestricted and Small-World underlay topologies, and about the same cost per-replica for the InterNAT and Erdős-Rényi topologies. We see the same ability of R^5N to create more replicas than R-Kademlia, again due to R^5N branching replication strategy. However, we are able to achieve similar levels of replication with our optimization, and achieve better performance than R-Kademlia at the same time. Another important note is that while stopping PUT requests when the first nearest peer is encountered is cheaper than forwarding up to $h = 2 \cdot T$ hops; both methods incur routing costs which increase linearly. It is important that none of our routing includes costs which increase at a greater rate.

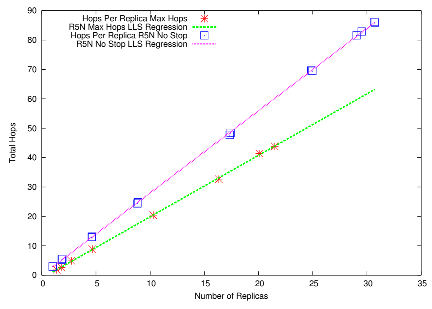
Figure 7.7 compares the total number of replicas present in the network after several rounds of PUT operations for the same key-value pair under the assumption that replicas persist indefinitely at a peer. The figure shows the number of replicas achieved by both R-Kademlia and R^5N for the case where either always the *same* peer performs the PUT operation or where the source



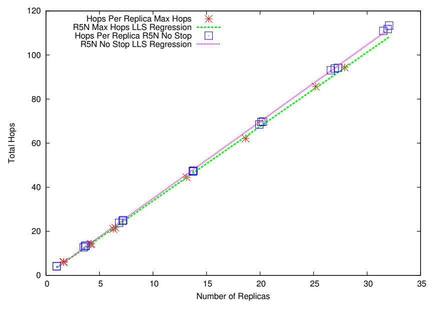
(a) Unrestricted



(b) 75% NATed



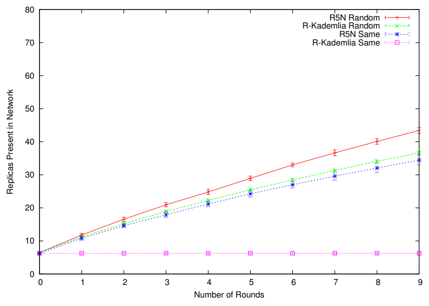
(c) Small-World



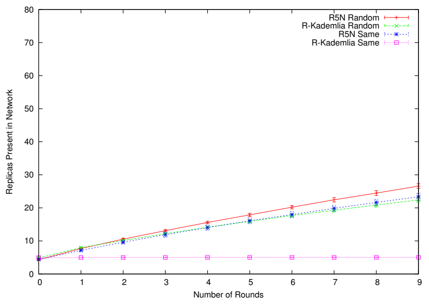
(d) Random Graph

Fig. 7.6: Hops required per replica; varying replication level r. These plots compare our normal forwarding behavior where requests terminate once they've reached $h > 2 \cdot T$ to forwarding where requests continue until they have reached a nearest peer.

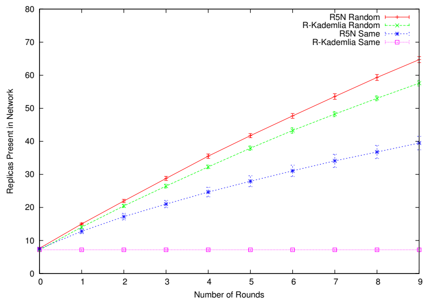
of the PUT operation is chosen at *random*. If the same peer performs the PUT operation using R-Kademlia, the PUT paths always converge at the same nearest peers and hence the number of replicas remains constant. In contrast, with R^5N, the random routing phase achieves significantly higher levels of replication over time. If PUT requests in R-Kademlia are started at a random peer, the resulting replication levels are only slightly higher, suggesting that the random routing phase in R^5N achieves its mixing goal (as described in Section 7.3.3).



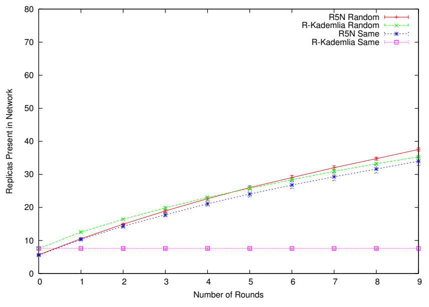
(a) Unrestricted



(b) 75% NATed



(c) Small-World

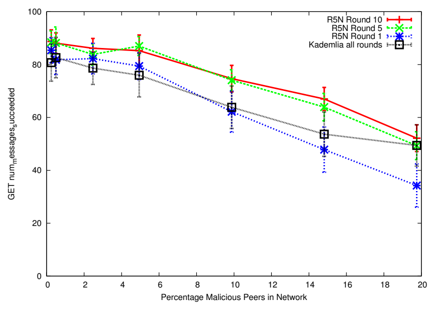


(d) Random Graph

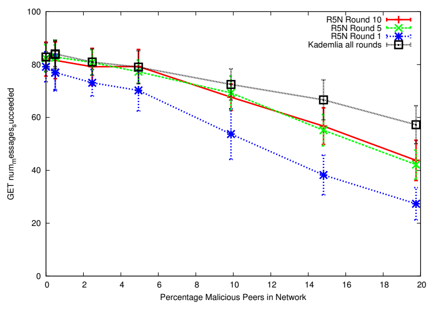
Fig. 7.7: Replication over time; same starting peer vs. randomized starting peers.

7.6.6 Malicious Peers

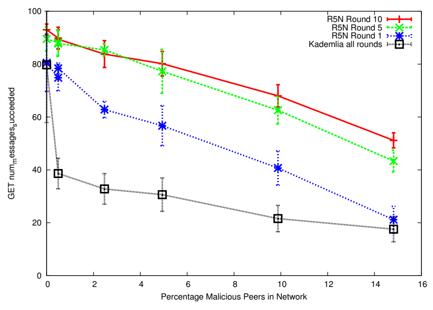
An additional goal for our routing algorithm is to perform well in the presence of malicious participants. Different types of attackers and attacks that could be considered are detailed in Section 7.3.8. We consider two types of malicious peers in our experimental analysis, the first being peers which join the network and simply passively drop all requests that are received. This type of adversary represents the simplest, and least detrimental kind of attack on any network. Peers may drop requests due to bandwidth restrictions, implementation bugs or naïve attacks. Thus this adversary does not target specific peers or identifiers in the network, and while not the most realistic type of attacker, it does provide insight into how the respective algorithms cope with possibly benevolent misbehaving peers. For deterministic algorithms, like R-Kademlia, this kind of attack is already quite detrimental to overall operation: any request that traverses any of the malicious peers fails. Redundancy (r-replication) and shorter paths compared to R^5N barely reduce the effectiveness of the attack.



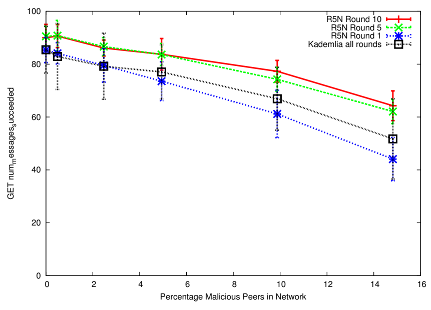
(a) Unrestricted



(b) 75% NATed



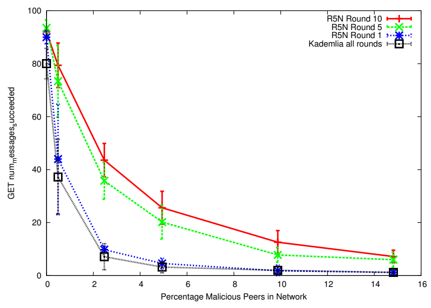
(c) Small-World



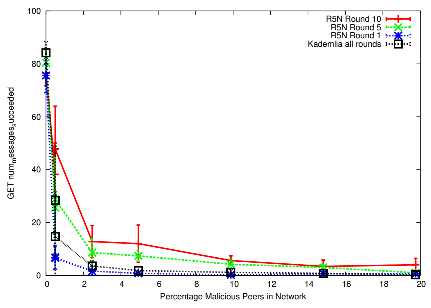
(d) Random Graph

Fig. 7.8: Percentage of malicious peers present at random locations in a network with 2025 peers vs. percentage of successful *GET* requests.

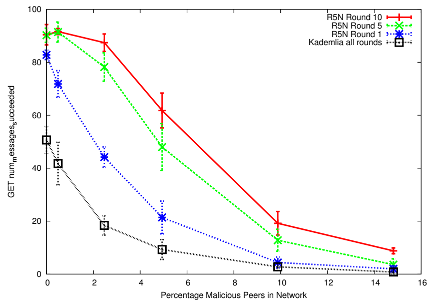
Figure 7.8 shows the impact of this type of adversary on the performance of the respective routing algorithms, for the case where the adversary does not attempt to eclipse a particular key but simply controls a random set of nodes and disrupts as many operations as possible (by dropping all requests). Figure 7.8 provides success rates for *GET* operations in a testbed with various topologies generated to have 30k edges. The lines represent the average success rate for *GET* operations initiated at peers selected uniformly at random in each round. These *GET* operations follow a number of rounds of *PUT* operations which are initiated at the same randomly chosen peer in each round. Later *GET* rounds in R^5N have higher success rates because additional rounds of *PUT* requests increase availability for R^5N as more replicas are inserted (as depicted in Figure 7.7). The benefit of R^5N over R-Kademlia is clearly seen in the randomized underlay topologies; while the unrestricted topologies are less affected by this type of attack for both R-Kademlia and R^5N.



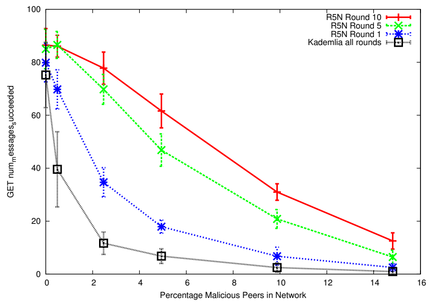
(a) Unrestricted



(b) 75% NATed



(c) Small-World



(d) Random Graph

Fig. 7.9: Percentage of malicious peers present at sybil locations in a network with 2025 peers vs. percentage of successful *GET* requests.

Finally, we consider an attacker controlling multiple nodes executing an Eclipse attack, that is, trying to prevent access to a particular key. The basic behavior of the malicious nodes is the same as in the previous case: they simply drop all *GET* and *PUT* requests. The key difference is that this time the μ malicious nodes are not placed into the network at random but at the μ peers that are also closest to the key. This represents an attacker able to perform a Sybil attack with free choice of identifier and node placement in a restricted-route topology — the strongest type of Sybil attacker we can imagine. This kind of attack is known to have a serious impact on Kademlia-based DHTs [170]. These Sybil peers do not "collude" in the strictest sense; as they do not actively coordinate during the attack. However, there is no benefit to such collusion, as we are modeling an attacker preventing access to a specific key. Colluding to forward requests to other known Sybil peers at each hop is not the defined goal of this attack.

R^5N does not have any explicit protections against Sybil attacks: the network is open and allows any peer to join without requiring an existing social network or certification by an authority. Furthermore, in our implementation, a peer's place in the key space is simply derived from the hash of its public key and hence an adversary can assign itself an identifier close to the desired key with relatively little effort. While additional protections against Sybil attacks could be added to the basic design, Figure 7.9 shows that they may often be unnecessary. As rounds of PUT requests increase the number of replicas in the network, R^5N yields much higher success rates. As expected, R^5N outperforms R-Kademlia; for R-Kademlia, a relatively small proportion of Sybil peers is able to drastically curtail performance, and repeating rounds of PUT and GET requests provides no improvement. Unsurprisingly, the worst results for both routing algorithms are seen in the unrestricted and InterNAT topologies. In these two topologies, routing tables are well populated, reducing the total number of nearest peers in the network and making the Sybils more effective at blocking access to the particular key being searched for. Also, R^5N is especially strong in the case of a Sybil attack on the Small-World underlay topology, where even the first round of GET requests succeeds with a much higher rate than R-Kademlia with only a few Sybil nodes.

We have shown that R^5N provides comparable or better replication (depending on topology) with the same or less cost than R-Kademlia. Thanks to this replication and R^5N unique routing strategy, R^5N is able to achieve performance on par with R-Kademlia in well connected topologies and vastly better performance in restricted-route topologies. Furthermore, the brief experimental results under both a very weak and very strong attacker scenario show that R^5N is able to cope with malicious participants in the network, especially in restricted-route networks where R-Kademlia is prone to failure. The next section provides a selection of other experimental data to further demonstrate the ability of R^5N under the very strong attacker, and at larger scale than the data we have presented so far.

7.7 Extended Data

This section contains interesting data which we collected during the course of this research over a time span of approximately 6 months using multiple desktop workstations, a 16 core server, and a 32 machine cluster. Most of the results in this section are presented showing networks with various topologies under the worst-case Sybil attack. We do this for two main reasons; first, data *without* malicious participants is basically a single data point[10] which is included on attack graphs as the area with 0 malicious participants. Second, the weak malicious "dropper" adversary is rather uninteresting (and an unlikely attack scenario), a very large proportion of peers must be dropping requests before the effect can be seen.

Another important aspect of the data we present here is effect of average node degree on R-Kademlia and R^5N routing. Our previous results assumed networks with a rather low total number of connections, around 30k in a network of size 2025. For the same small scale network size, we vary the number of connections to demonstrate how well R^5N and R-Kademlia operate with varying node degrees in the underlay and overlay topologies. These results are shown in Section 7.7.1.

A very important aspect of DHTs in particular, and P2P networks in general, is scalability. Thus far we have made the claim that R^5N should be scalable under the assumption that because R^5N uses Kademlia-style routing and routing tables, R^5N should scale as well as Kademlia does. However, it is always important to verify this via experimental results. This is the topic of Section 7.7.2.

Furthermore, for all of the various topologies (even those where the overlay topology is structured via *FIND PEER* requests), the *exact* same topology is used for all trials that are shown in a figure. This is accomplished via the topology capture and recreation functionality enabled by our emulation framework as described in Chapter 6. As such, the results only reflect differences in the routing algorithms themselves, and not variations of topology setup. Finally, each data point on each graph is an average of at *minimum* five trials, and includes standard deviations whenever relevant.

7.7.1 Small Scale Results

The majority of tests that we ran over the course of our evaluation of R^5N were at the same scale as those results presented previously, 2025 peers in various topologies. There are many different options we could specify for these tests, as detailed in Chapter 6, most of which we tested independently under specific scenarios to arrive at the parameters we then used in our more extensive testing. For instance, as discussed in Section 7.6, we fixed the values of $c = T = \log n$ and $r = 10$ in order to enable a fair comparison

[10] albeit for both routing algorithms with standard deviations

between R^5N and R-Kademlia. The data shown in this section follows this convention as well, unless otherwise explicitly noted as a variable which is under examination for the specific section.

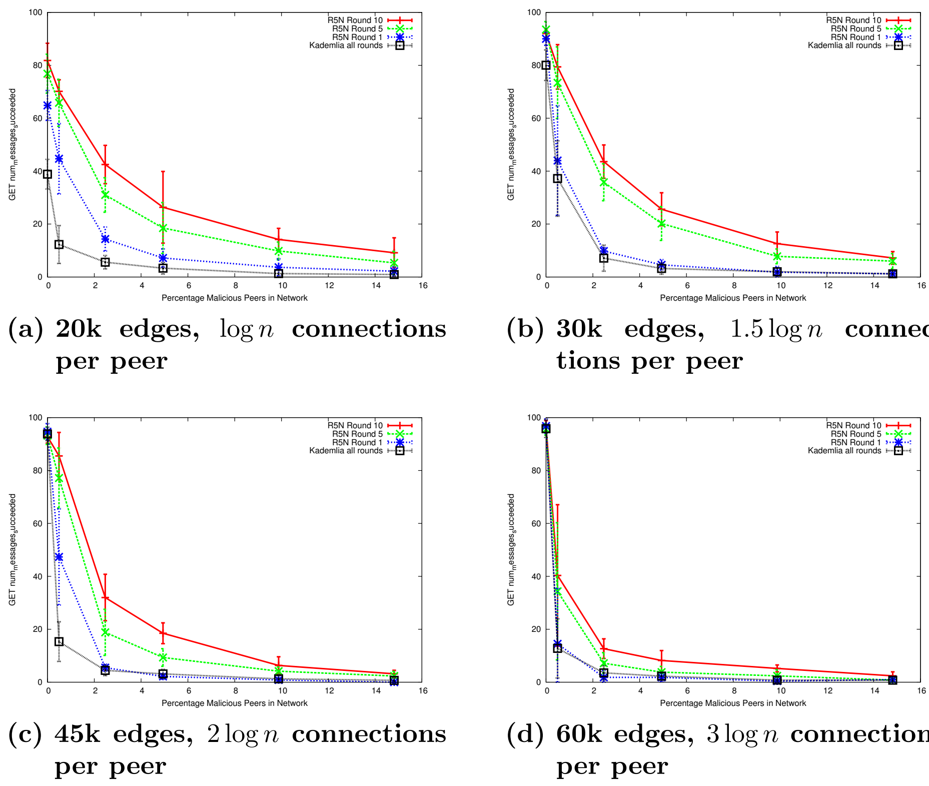
7.7.1.1 Unrestricted Topology with Sybil attack – Varying α

In this section, we view the effects of the total number of connections on routing performance in an unrestricted topology with Sybil attackers. The topology construction for these trials is rather unique; we allowed an unrestricted underlay topology, and allowed all peers to send *FIND PEER* requests up to the point that the total number of desired connections is reached. The intuition behind this topology construction is that networks with high rates of churn or where many new peers enter the network may be unrestricted at the underlay level, but not well connected because all peers in the network have not been around long enough for routing tables to converge. This is actually a likely scenario, given that multiple rounds of *FIND PEER* requests are required for convergence even in a static topology.

We would expect that, ignoring the effects of Sybil attackers in the network, as more connections are allowed in the final topology both R-Kademlia and R^5N should show increased success in requests. This is precisely the impact we see in the results shown in Figure 7.10; in Figure 7.10a R-Kademlia achieves only a 40 percent success rate without attackers, which increases up to nearly 100 percent in Figures 7.10c and 7.10d. R^5N achieves a higher success rate than R-Kademlia in the least well connected topology shown in Figure 7.10a, reaching up to 80 percent of request success in the 10^{th} round, compared to R-Kademlia's 40 percent. With a higher level of connectivity, both R-Kademlia and R^5N are comparable without the presence of Sybil peers.

When taking the Sybil attack peers into account, we can see that in all of these unrestricted topologies performance decreases drastically as more Sybil peers are added to the network. This is to be expected, as the number of nearest peers decreases as routing tables become more sufficiently populated. Encouragingly, R^5N does achieve better performance in the 5^{th} and 10^{th} rounds, even when there are many Sybil attackers present in the network. The ability of R^5N to combat the Sybil attack decreases as the level of connectivity increases, again because the number of replicas in the network gets too low for R^5N to replicate at a sufficient number of peers.

The result from this data is that R^5N may be a suitable choice for networks with very high rates of churn, or those that are restricted to approximately $\log n$ total connections at any one time. Without malicious peers, R^5N is able to achieve good performance even while a network is converging via *FIND PEER* requests to the optimal overlay topology. Also, networks with Sybil attackers may benefit from using R^5N as well, more so if the network also happens to have high churn rates.



(a) **20k edges,** $\log n$ **connections per peer**

(b) **30k edges,** $1.5 \log n$ **connections per peer**

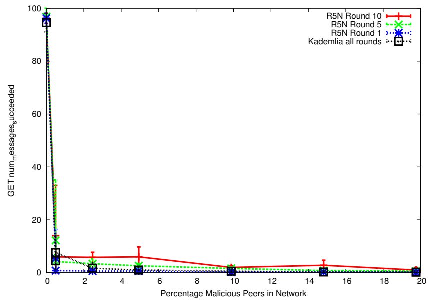
(c) **45k edges,** $2 \log n$ **connections per peer**

(d) **60k edges,** $3 \log n$ **connections per peer**

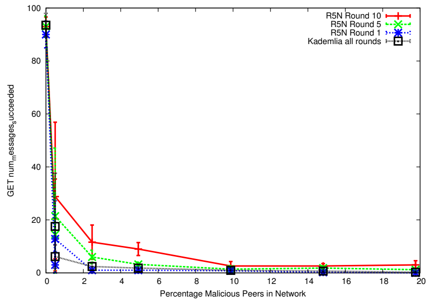
Fig. 7.10: Sybil attack in 2025 peer network with unrestricted topology; varying α. Kademlia and R^5N are comparable in the first round; R^5N has higher success rates in later rounds. *GET* requests were started at randomly chosen peers in each round, *PUT* requests were initiated from the same randomly chosen peer in each round.

7.7.1.2 InterNAT Topology with Sybil Attack – Varying α

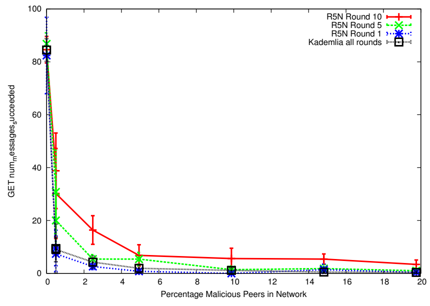
In this section, we analyze the total number of connections in an InterNAT topology on R^5N and R-Kademlia routing. Incorporated into this analysis are the effects Sybil attackers on those topologies. To create these topologies, we imposed the NAT restrictions of a certain percentage of peers that were unrestricted; the remaining peers could only connect in the underlay to those unrestricted peers. We then had peers send *FIND PEER* messages until the topology stabilized (sending more *FIND PEER* requests no longer increased the number of connections). When restricting 50 percent or less peers in this way, the results are virtually identical to those for a completely unrestricted topology. This is because there are more than enough peers to populate Kademlia routing tables, the only effect is that buckets for half of the peers are slightly less full. Therefore, we show results from various levels of NAT restriction upwards from 50 percent of peers NATed.



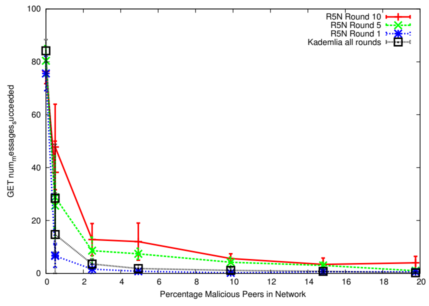
(a) 50% NATed, 75k edges, $3\frac{1}{2}\log n$ connections per peer



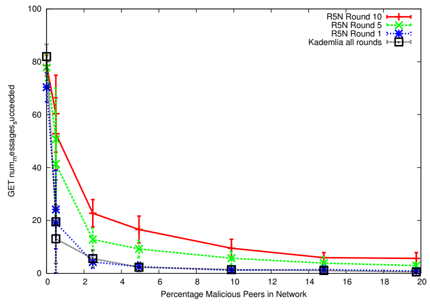
(b) 65% NATed, 60k edges, $3\log n$ connections per peer



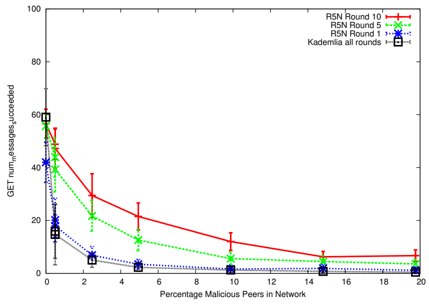
(c) 70% NATed, 52k edges, $2\frac{1}{2}\log n$ connections per peer



(d) 75% NATed, 45k edges, $2\log n$ connections per peer



(e) 80% NATed, 36k edges, $1\frac{1}{2}\log n$ connections per peer



(f) 90% NATed, 20k edges, $\log n$ connections per peer

Fig. 7.11: Small scale 2025 peer network, Sybil peers, NAT restricted topologies, varying percentages of peers NATed. All *PUT* requests were initiated at the same randomly chosen peers, while *GET* requests are performed at random starting peers in each round.

Figure 7.11 shows results from six of these NATed topologies, ranging from 50 to 90 percent of peers NATed. A first interesting result of these topologies is the number of total connections in the network, which decreases significantly as more and more peers are restricted. In Figure 7.11f, there are only 20k total connections in the network. While we give a per-peer estimate of connections in each of the figures, it should be noted that the effect of the NAT restrictions means that the distribution of connections is not spread out equally over the peers. The unrestricted peers generally have many more connections in their (properly filled) routing tables, and the NATed peers have few connections.

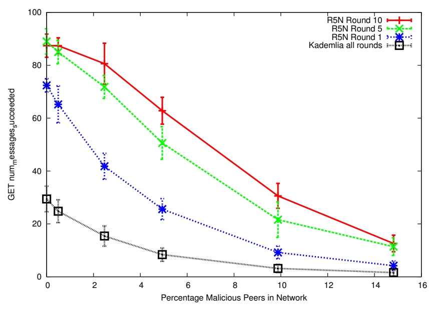
Without malicious peers, we see that R-Kademlia and R^5N perform comparably in these NAT topologies. R-Kademlia seems to have a slightly higher success rate than R^5N, especially in the first round of R^5N *GET* requests. Both algorithms fare worse as the percentage of NATed peers increase, without considering Sybil peers.

The Sybil attack is clearly particularly devastating on this topology, both because of the high level of connectivity (with the lower NAT percentages) and because of the lopsided distribution of connections into the routing tables of unrestricted peers. Even so, R^5N still achieves higher rates of success after 5 and 10 rounds of PUT requests than R-Kademlia. Clearly, for the networks with more NATed peers the increase in performance of R^5N in subsequent rounds becomes more pronounced; however, the baseline success rate starts out lower. The conclusion we make here is that for InterNAT topologies, neither routing algorithm works particularly well for a network under Sybil attack, but R^5N does achieve improvement, especially in highly restricted topologies.

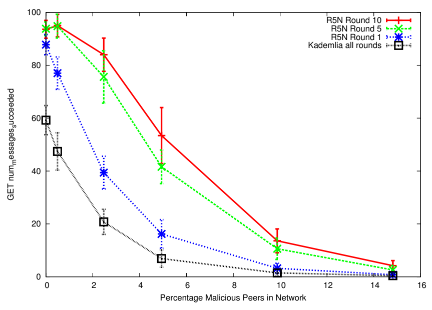
7.7.1.3 Small-World Topology with Sybil attack – Varying α

This section details the effects of Sybil attacks on Small-World networks with varying degrees of connectivity in the underlay topology. Specifically, a parameter which controls the average node degree in the underlay topology generator is adjusted to create the desired topology. In our results thus far, Small-World networks have shown the best performance for R^5N. Of course, this is exactly as we had expected; the short paths between peers enables our random routing phase to find sufficiently randomized peers in the network. The relatively low average node degree in these networks means that there are a large number of nearest peers to any random key. This allows us to achieve our required level of replication, and thus provide better performance in the network. The question is, at what point (insofar as the average connectivity in the topology) do the advantages for R^5N start to disappear?

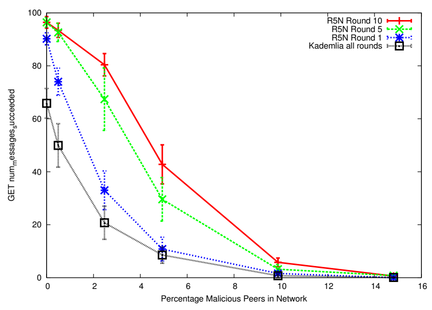
Figure 7.12 shows results from Small-World topologies ranging from average per-peer connections of $\log n$ in Figure 7.12a up to $4 \log n$ in Figure 7.12f. We believe this covers almost the entire range of likely Small-World networks; higher levels of connectivity give clique-like performance in routing.



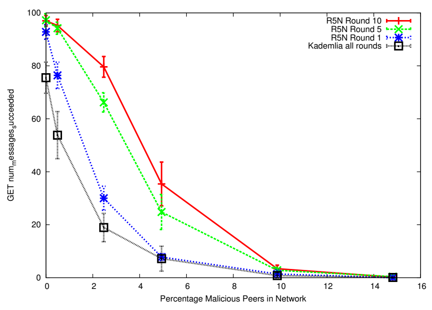
(a) 25k edges, $\log n$ connections per peer



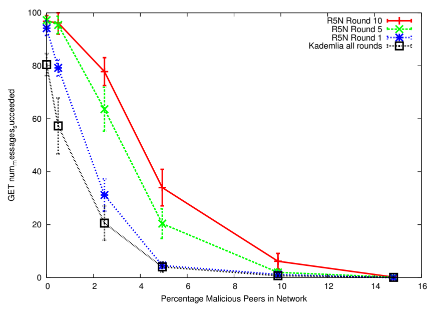
(b) 45k edges, $2 \log n$ connections per peer



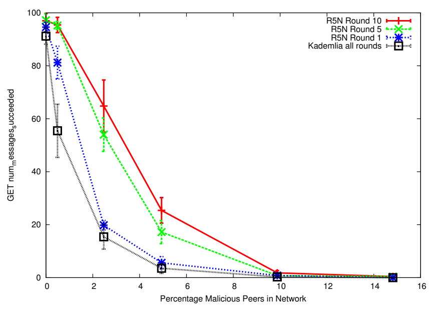
(c) 56k edges, $2.5 \log n$ connections per peer



(d) 65k edges, $3 \log n$ connections per peer



(e) 73k edges, $3.5 \log n$ connections per peer



(f) 85k edges, $4 \log n$ connections per peer

Fig. 7.12: Sybil attacks in 2025 peer networks, Small-World topologies with varying α. R^5N outperforms Kademlia consistently, regardless of the number of Sybil peers; R^5N also has higher success rates in later rounds. *GET* requests were started at randomly chosen peers in each round, *PUT* requests were initiated from the same randomly chosen peer in each round.

Ignoring the impact of the Sybil attack for the time being, we can see that R-Kademlia has poor performance in all but the very dense topologies.

Figure 7.12a shows a success rate of less than 30 percent for R-Kademlia without malicious peers, whereas R^5N has close to a 70 percent success rate in the first round, and up to 90 percent success in later rounds. R-Kademlia performance does not reach the level of R^5N in these topologies until there are 85k total connections, as shown in Figure 7.12f.

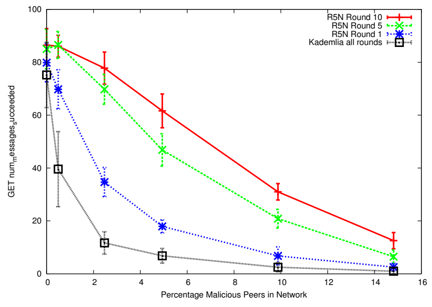
R^5N truly shines when we look at the results of the Sybil attack on the network in these Small-World topologies. The previously discussed results for unrestricted (Section 7.7.1.1) and InterNAT (Section 7.7.1.2) showed only small improvement for R^5N over R-Kademlia when under Sybil attack. Figure 7.12 shows that the improvement is more pronounced for Small-World topologies. While in the first round, the R^5N results are close to that of R-Kademlia, the 5^{th} and 10^{th} rounds show marked improvement over R-Kademlia. Perhaps the most drastic example is shown in Figure 7.12a where R^5N in the 10^{th} rounds consistently has 4 times the success rate of R-Kademlia. Comparing the overall shape of the R^5N curves from Figure 7.12 to Figures 7.10 and 7.11 reveals that with R^5N the drop-off as more Sybil peers are added is less pronounced. This indicates that in a Small-World topology, R^5N is a particularly good choice over Kademlia for networks which may need to withstand Sybil attacks.

7.7.1.4 Erdős-Rényi Topology with Sybil attack – Varying α

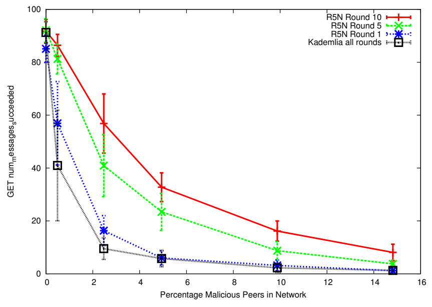
In this section, we show the impact of the total number of connections in an Erdős-Rényi topology on R^5N and R-Kademlia routing while also examining the effects of malicious Sybils. In general, for random topologies with a greater number of connections we expect that R-Kademlia will achieve higher success rates than in topologies with fewer connections. Obviously, Kademlia routing tables will be more full, there will be fewer nearest peers in the network, and R-Kademlia should be able to find a sufficient number of nearest peers to ensure success.

Figure 7.13 shows the results from Erdős-Rényi topologies with varying average connections from $\log n$ (Figure 7.13a) to approximately $2\frac{1}{2}\log n$ (Figure 7.13d). The results for R-Kademlia are as we expected; with only $\log n$ connections per peer, we see in Figure 7.13a that less than 80 percent of requests are successful for R-Kademlia. As the average number of connections increases, we see rapid improvement in R-Kademlia success rates up to nearly 100 percent in Figure 7.13d. Without considering attackers, R^5N follows the same trend, achieving around an 85 percent success rate with $\log n$ average connections and nearly 100 percent with $2\frac{1}{2}\log n$ connections.

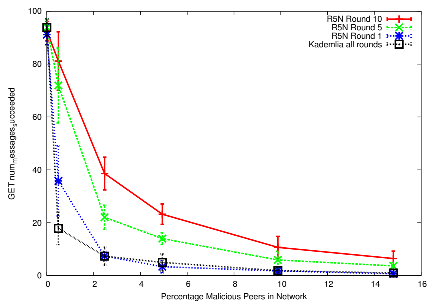
The similarity between R^5N and R-Kademlia ends upon consideration of the malicious Sybils in the network. In all of the four Erdős-Rényi topologies shown in Figure 7.13, regardless of the total number of connections, R^5N outperforms R-Kademlia. R^5N also has continued improvement in the 5^{th} and 10^{th} rounds, while R-Kademlia of course cannot improve in these subse-



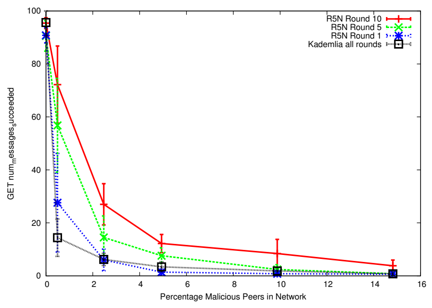
(a) 20k edges, $\log n$ connections per peer



(b) 30k edges, $1\frac{1}{2}\log n$ connections per peer



(c) 40k edges, $2\log n$ connections per peer



(d) 55k edges, $2\frac{1}{2}\log n$ connections per peer

Fig. 7.13: *GET* requests were started at randomly chosen peers in each round, *PUT* requests were initiated from the same randomly chosen peer in each round.

quent rounds. We also see what will become a trend in many of our various topologies; as the number of connections per peer increases, the Sybil attack becomes more and more effective against R^5N. This allows less well connected topologies to provide *better* performance than better connected topologies under a Sybil attack with specific number of Sybil peers present.

This concludes what we believe to be the most interesting results from our small scale testing. For the most part, the outcome of the trials are what we expected. Specifically, in well connected topologies (greater than approximately $3\log n$ connections per peer), R-Kademlia has enough connections to perform efficient routing. However, R-Kademlia is incredibly vulnerable to Sybil attacks, with the success rates plummeting to nearly 0 with a small number of attacking peers in the network. R^5N achieves high success rates in both well connected and sparse topologies, and gives some protection against Sybil attacks; certainly better than that provided by R-

Kademlia. R^5N performs particularly well in Small-World topologies due to their unique properties combined with R^5N's replication strategy and randomized routing design.

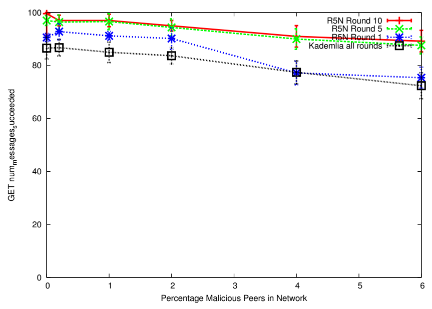
7.7.2 Large Scale Tests

In addition to running small scale tests, which show that R^5N performs as we expected in myriad topologies and attacker scenarios, we would also like to know that R^5N is able to scale as well as Kademlia. This requires testing R^5N in larger networks of peers. However, while our emulation framework is scalable (as discussed in Chapter 6), larger scale tests invariably take more time to execute than small scale tests. The bottleneck for emulations is typically the total number of connections in the network, as connecting peers into a desired topology is the most time consuming task the framework must perform. As an example, a 2025 peer emulation including all phases of DHT testing from peer startup to peer shutdown can run on a single desktop in about 30 minutes. The same test with 40,000 peers requires use of the full cluster at our disposal of 32 hosts and takes about 3 hours.

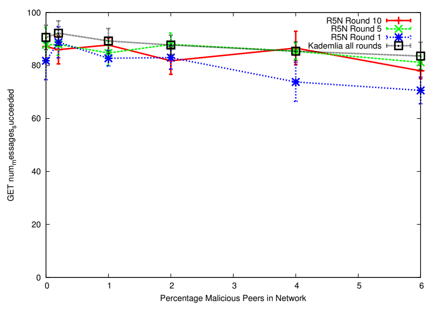
Furthermore, at large scale, it is unrealistic to gather full routing details including every hop of every message. Therefore, in our large scale tests, we limit results to the highest level, specifically the number of *GET* requests that are ultimately successful. For these reasons, we have limited our large scale testing to those scenarios which we thought were most interesting, or where the small scale results were surprising. To test both R^5N and R-Kademlia exhaustively in all scenarios would simply take too much time, and waste the resources of CPU time and energy on tests with obvious results. One notably absent topology from *all* of the large scale tests is the unrestricted clique topology. There are two reasons for this; first, due to operating system limitation it is simply impossible to achieve the number of connections required for even "small" networks of approximately 2,000 peers. Second, we expect that R-Kademlia will always outperform R^5N in these topologies as far as efficiency is concerned. Also, as our GNUnet DHT implementation requires many nearest peers in order to achieve replication, topologies with few (1, in the worst case) nearest peers and Sybil attackers will perform disastrously (as we saw in Section 7.7.1.1). Again, these topologies are of limited interest to us at this time, as our focus is on restricted route topologies with many replicas for any given random identifier.

7.7.2.1 5,000 Peers

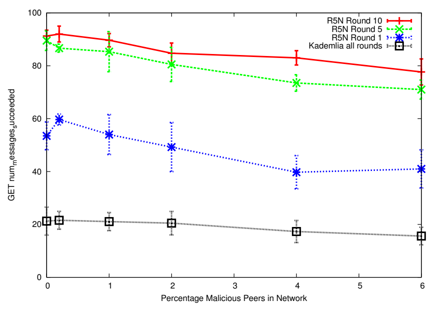
Our small scale tests were run with 2,025 peers, for the reasons described earlier in this chapter. Since we expect performance to degrade at a logarithmic pace, we increase the network size exponentially, starting with 5,000. This is obviously more than twice 2,025, but since 2,025 peers was a rather peculiar number in the first place, we decided to use a more obvious choice for larger scale tests. Thus, in this section we show interesting results from a 5,000 peer network.



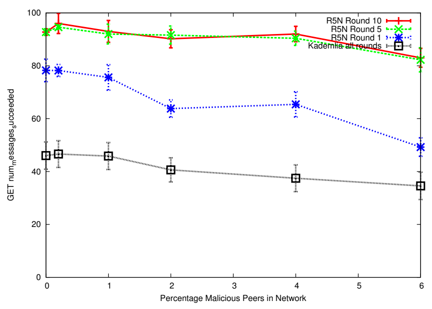
(a) Erdős-Rényi topology; 283k edges, $4\frac{1}{2}\log n$ connections per peer



(b) NAT topology; 130k edges, $2\log n$ connections per peer



(c) Small-World topology; 80k edges, $\log n$ connections per peer



(d) Small-World topology; 160k edges, $2\frac{1}{2}\log n$ connections per peer

Fig. 7.14: Results from interesting 5,000 peer tests with malicious droppers. *PUT* requests were initiated from the same randomly chosen peer in each round, *GET* requests were started at randomly chosen peers in each round.

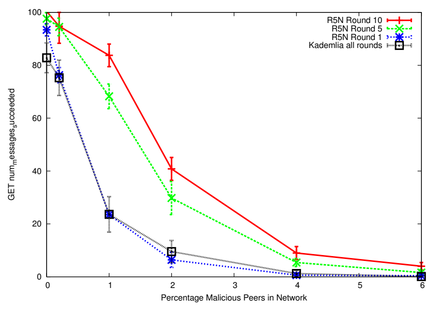
Figure 7.14 shows a selection of results from multiple topologies with the inclusion of malicious dropper peers. An obvious, and expected, result is that the inclusion of malicious droppers has little effect on routing in any of the topologies. This stands to reason due to the large number of nearest peers in the network, and the small number of "dumb" malicious peers which only drop requests. This does however bolster the small scale result that mis-configured peers or those unable to participate in the DHT should have little impact on the overall success of queries. An unexpected result is shown in Figure 7.14a, which shows a well connected ($4 \cdot \log n$ connections per peer) random topology. In the small scale tests, such well connected random topologies showed R-Kademlia outperforming R^5N. However, as we increase the total number of peers in the network, this effect seems to decrease. This is likely because even with only twice as many peers, the number of connections possible in the network increases by a factor of 6. Thus, the required connections to enable R-Kademlia routing are less likely to be made.

In contrast to this result, with the InterNAT topology at the same scale, even with fewer total connections ($2 \cdot \log n$ per peer) we see the same result that we saw in small scale tests. In particular, Figure 7.14b shows R-Kademlia consistently outperforming R^5N. This is because while there are fewer connections allowed, the InterNAT topology still allows peers to choose which connections to even attempt (via *FIND PEER* requests). Therefore the routing tables achieve better diversity than with completely random connections, as in Figure 7.14a.

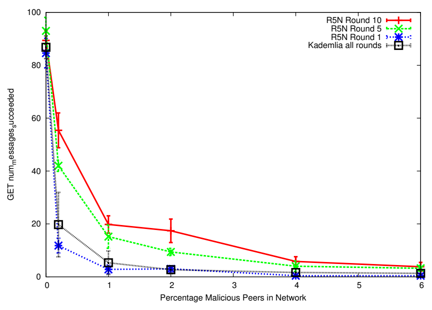
The best results for R^5N are seen in Figures 7.14c and 7.14d. These also show the same number of malicious dropping peers, only this time connected in Small-World topologies with varying average total connections per peer. Figure 7.14c shows a Small-World topology with on average only $\log n$ connections per peer. R-Kademlia is only able to achieve roughly a 20 percent success rate in this sparse topology, R^5N achieves approximately twice this success rate even with only a single round of *PUT* requests issued. However, the real benefit of R^5N is seen in rounds 5 and 10, where approximately 90 percent of *GET* requests succeed without malicious peers, and this only decreases to around 80 percent in the presence of 300 malicious peers. Figure 7.14d shows a Small-World topology with double the total connections as Figure 7.14c. For R-Kademlia, this equates to a doubling of the initial success rate from Figure 7.14d to about 40 percent; R^5N sees less improvement, with between 60 and 80 percent success rates in the first round. However, in rounds 5 and 10, R^5N achieves over 90 percent success for *GET* requests, up to 200 malicious droppers in the network. The clear result of these tests are that R^5N clearly outperforms R-Kademlia in a 5,000 peer Small-World network, whether there are peers dropping requests or all peers are behaving properly. The results for the Erdős-Rényi topology are different from the small scale tests, but unsurprisingly.

Figure 7.15 shows results of a Sybil attack on the 5,000 peer network. As would be expected, the more randomized the topology, the better the defense against the Sybil attack. As explained with Figure 7.14, even though the Erdős-Rényi topology has a greater number of connections, it is less suited to Kademlia style routing. Therefore, comparing between Figure 7.15a and Figure 7.15b, we see that the Erdős-Rényi topology better resists the Sybil attack. However, in both the Erdős-Rényi and InterNAT topologies, the Sybil attack remains quite detrimental to overall *GET* request success rates.

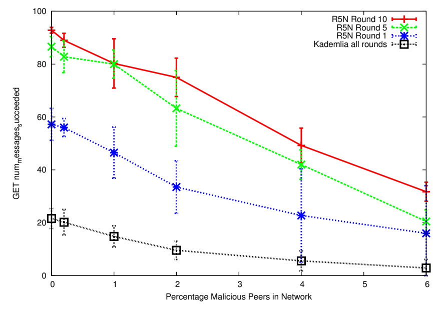
The impact of the Sybil attack on the network topologies shown in Figures 7.15a and 7.15b is sharply contrasted with the results in Figures 7.15c and 7.15d. In both of these figures, the performance of the network suffers



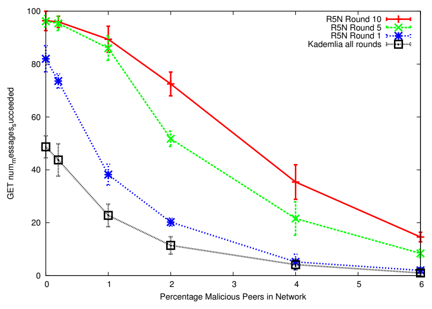
(a) Erdős-Rényi topology; 283k edges, $4\frac{1}{2}\log n$ connections per peer)



(b) NAT topology; 130k edges, $2\log n$ connections per peer



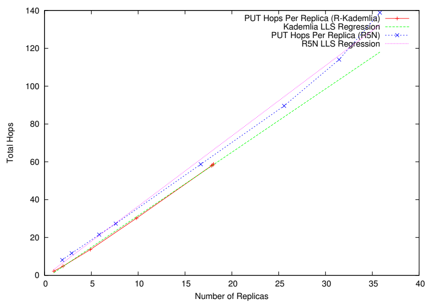
(c) Small-World topology; 80k edges, $\log n$ connections per peer



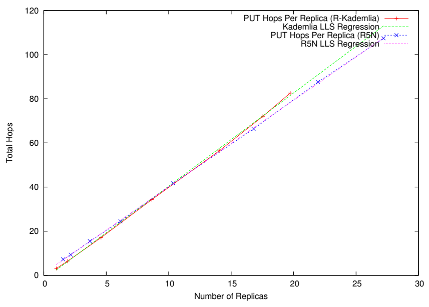
(d) Small-World topology; 160k edges, $2\frac{1}{2}\log n$ connections per peer

Fig. 7.15: Results from interesting 5,000 peer tests with malicious Sybil peers. *PUT* requests were initiated from the same randomly chosen peer in each round, *GET* requests were started at randomly chosen peers in each round.

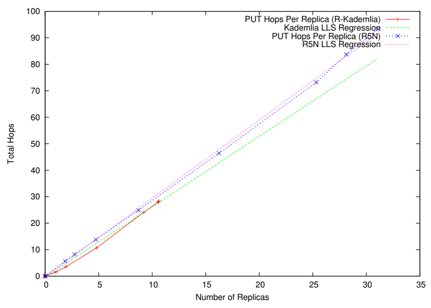
significantly less from the addition of Sybil peers. As with the malicious dropper tests, R^5N outperforms R-Kademlia even in the first round. This improvement is further increased in the 5^{th} and 10^{th} rounds. While the Sybil attacks remain detrimental to the network regardless of routing algorithm, it is clear that R^5N may be usable even in the presence of a large number of Sybil attackers. Another aspect of the Small-World topologies that is revealed by comparing Figures 7.15c and 7.15d is the trade-off between the initial performance of the network (without malicious peers) and the impact of malicious peers on the network. Specifically, in the better connected topology seen in Figure 7.15d the initial success rate is higher than in Figure 7.15c, but success rates decrease faster as Sybil peers are added, with a success rate roughly half that of Figure 7.15c with 300 malicious Sybil peers (the maximum shown) in the network. The conclusion from this is that better performance may be achieved in sparse networks under Sybil attack than densely connected topologies.



(a) Erdős-Rényi topology; 283k edges, $4 \log n$ connections per peer



(b) InterNAT topology; 130k edges, $2 \log n$ connections per peer



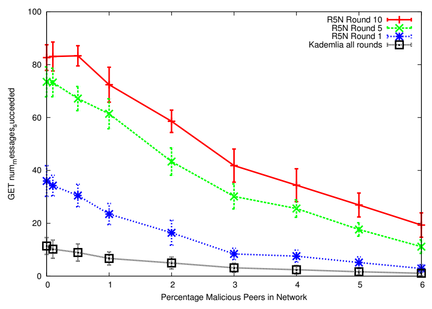
(c) Small-World topology; 80k edges, $\log n$ connections per peer

Fig. 7.16: Average hops required per replica; varying replication level r.

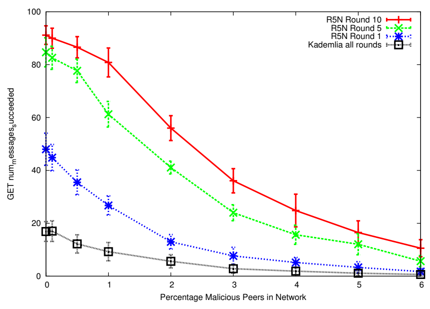
Figure 7.16 shows the hops required per replica for 5,000 peer topologies given in this section, in the same manner as shown in Figure 7.5. The comparison is again between R^5N and R-Kademlia, and plots the average total hops required for each nearest peer found for *PUT* requests. The x-axis plots the total number nearest peers reached on average for a round of *PUT* requests. We plot this against the total number of hops required on average on the y-axis. The results are virtually identical to the smaller scale tests; both algorithms are very similar in the total hops required per replica, and both seem to increase linearly. Linearity is important so that we ensure that achieving the *next* replica is not more costly than the last. The same problem with R-Kademlia is present in these larger scale tests as well, that R-Kademlia is only able to find nearest peers up to a certain point due to the average number of connections and convergence of multiple paths to the same nearest peer. Even in the well connected Erdős-Rényi topology, R-Kademlia can reach less than 20 nearest peers. In the Small-World topology, R-Kademlia can only reach around 10 nearest peers. R-Kademlia performs best in the InterNAT topology (due to it being the most highly structured overlay topology), reaching up to 20 nearest peers. Conversely, R^5N performs *worst* in this topology, as we would expect. However, in *all* topologies, R^5N is able to reach a greater total number of nearest peers, showing that routing randomization can achieve a higher number of replicas in virtually any restricted-route topology than R-Kademlia is able to. This higher number of nearest peers can be found without decreasing performance, seen in the similarity of the number of hops required for the two algorithms.

7.7.2.2 10,000 Peers

We now double the total number of peers in the topologies from 5,000 to 10,000. Due to the time required to run these tests as outlined in the beginning of this section, we limit these results to Small-World topologies. We believe that the InterNAT topologies can be safely assumed to continue to outperform R^5N due to their closeness to completely unrestricted topologies. We also believe that higher levels of replication can still be achieved using R^5N even in InterNAT topologies where only a portion of users are restricted. These increased levels of replication can increase resilience to failures and malicious peers in the form of dropping or Sybil attacks. Nonetheless, constructing InterNAT topologies at larger scales is increasingly difficult due to the large size of blacklists and whitelists required to be read into memory at each peer. As such, we continue our results with tests on randomized Small-World topologies.



(a) **Small-World topology, 180k edges, $\log n$ connections per peer**



(b) **Small-World topology, 280k edges, $2 \log n$ connections per peer**

Fig. 7.17: Results from Small-World topologies with malicious Sybil peers. *PUT* requests were initiated from the same randomly chosen peer in each round, *GET* requests were started at randomly chosen peers in each round.

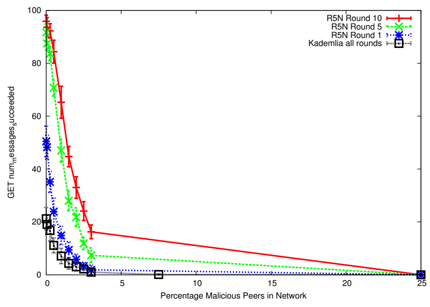
The plots in Figure 7.17 show a 10,000 peer network with approximately $\log n$ and $2 \log n$ connections per peer. Results are plotted with 0 to 600 Sybil attackers present in the network. The results follow almost exactly what we would expect after seeing the small scale and 5,000 peer results. We see that R-Kademlia achieves very low success rates, around 10 percent in Figure 7.17a and 20 percent in Figure 7.17b. This is roughly half the rate of success achieved in Figure 7.15, a 5,000 peer network with the same proportion of connections. Again, R^5N achieves nearly twice the success rate of R-Kademlia, even after only a single round of *PUT* requests. R^5N continues to improve with more rounds of *PUT* requests, achieving success rates around 80 percent in Figure 7.17a and over 90 percent in Figure 7.17b, when there are no Sybil attackers. Again, the Small-World network with more connections achieves better initial performance, but is also more susceptible to the Sybil attack, with Figure 7.17a showing a higher success rate when more than 2 percent (200) of the peers in the network are Sybil attackers.

7.7.2.3 20,000 Peers

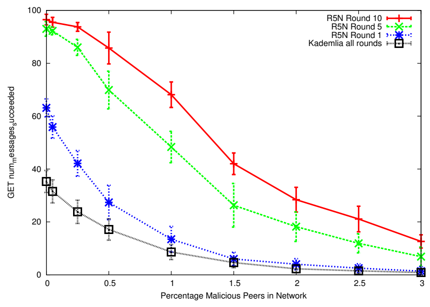
We again double the number of peers we are emulating for a total of 20,000 peers. As with the 10,000 peer tests, we concentrate on random topologies. For these tests, we concentrate on well connected random topologies, with approximately $3 \log n$ connections per peer. As observed in the small scale tests, with this many connections in the Erdős-Rényi topology R-Kademlia was able to beat R^5N in success rates of *GET* requests when there were no malicious peers participating in the network. Therefore it is interesting to see if this result changes as the scale increases.

Figure 7.18a shows, as was the case with 10,000 peers in the Small-World topology, R-Kademlia is unable to match R^5N performance with or without malicious peers. In this topology, R^5N has about $2\frac{1}{2}$ times the R-Kademlia success rate of approximately 21 percent in round 1, and between 4 and 5 times this success after 5 and 10 rounds (close to 100% success). Increasing the number of Sybil peers reduces the performance for both algorithms. This figure also shows what happens when we increase the number of Sybil peers beyond the total number of replicas in the network (to 5,000 malicious Sybil peers); specifically, all requests fail. This somewhat extreme data point is meant to reinforce the fact that with enough Sybil peers in a network, no routing algorithm will be able to cope.

Figure 7.18b shows results from a Sybil attack on the same network size, this time with peers connected in a Erdős-Rényi topology with roughly the same number of connections as the Small-World topology depicted in Figure 7.18a. This data is quite encouraging for R^5N; it seems to indicate that as the network size increases R^5N performs increasingly well in Erdős-Rényi topologies as compared to R-Kademlia. Comparing this result with



(a) **Small-World topology, 920k edges, $3 \log n$ connections per peer**



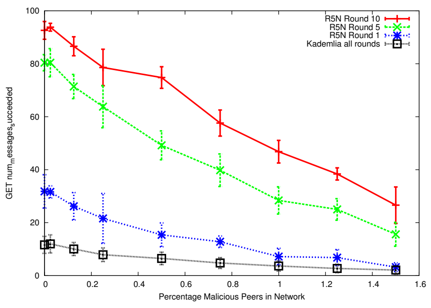
(b) **Erdős-Rényi topology; 870k edges, $3 \log n$ connections per peer**

Fig. 7.18: Sybil attack in 20,000 peer networks for Small-World and Erdős-Rényi topologies with approximately $3 \log n$ connections each. In these topologies, R^5N outperforms R-Kademlia across the board. *GET* requests were started at randomly chosen peers in each round, *PUT* requests were initiated from the same randomly chosen peer in each round.

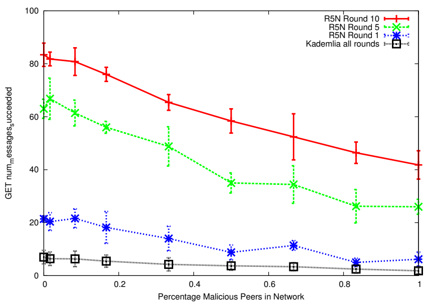
that seen in the 5,000 peer Erdős-Rényi topology from Figure 7.14a shows that while R-Kademlia drops to less than half the success rate, R^5N remains at nearly 100 percent success with no malicious Sybil peers. Furthermore, in later rounds with more Sybil peers, R^5N shows great improvement over R-Kademlia. With 500 Sybil peers in the network, R-Kademlia achieves less than 1 percent of successful requests, whereas R^5N has over 20 percent successful.

7.7.2.4 More than 20,000 Peers

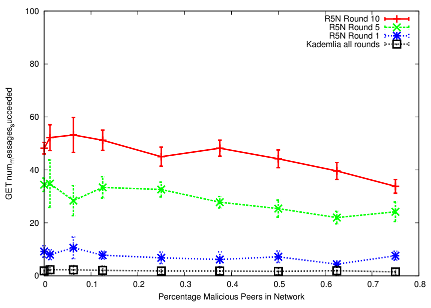
As discussed in Chapter 6, the largest emulations we have found in previous work have been around 4,000 peers. We consider our results up to 20,000 peers to be quite an achievement, running almost 5 times the number of peers that emulations were capable of previously. We believe that performing well up to 20,000 peers also makes our design usable on real-world networks, as few secure P2P networks have thus far made it to this scale. Furthermore, our implementation is immediately ready for real world deployment. However, we were also curious to see just how many peers we could run using our emulation framework on the cluster we had at our disposal. This section outlines some of the data we gathered while testing the bounds of this framework. As in previous sections, time constraints on the cluster and our focus on Small-World networks restricted our tests to specific topologies. The topology we chose to focus these tests on is a Small-World topology with approximately $2.5 \log n$ connections per peer. This comes out to around 50 connections per peer, depending on the exact topology. We believe this number of connections to be realistic for P2P users who may have limited bandwidth or connection restrictions.



(a) Small-World topology, 40,000 peers, Sybils, 2,000,000 edges, $3 \log n$ connections per peer



(b) Small-World topology, 60,000 peers, Sybils, 2,600,000 edges, $3 \log n$ connections per peer



(c) Small-World topology, 80000 peers, Sybils, 2,000,000 edges, $2 \log n$ connections per peer

Fig. 7.19: Plots for data from tests when running more than 20,000 peers. *GET* requests were started at randomly chosen peers in each round, *PUT* requests were initiated from the same randomly chosen peer in each round.

Figure 7.19 shows tests we performed on network sizes of 40,000 (Figure 7.19a), 60,000 (Figure 7.19b) and 80,000 (Figure 7.19c). Our intended number of per peer connections was approximately $2\frac{1}{2} \log n$, though in the tests for Figure 7.19c we were only able to achieve roughly $2 \log n$ (due to system limits).

These figures echo the results of small scale testing; in Small-World topologies R^5N is able to provide a significantly higher success than R-Kademlia, with or without malicious participants in the network. Indeed, as the scale increases upwards of 40,000 peers, R-Kademlia becomes almost completely ineffective. On the other hand, R^5N provides reasonable levels of success, even when under the powerful Sybil attack. As before, Figure 7.19 shows that in the first round R^5N has similar (though slightly better) performance than R-Kademlia; though this performance is vastly improved after multiple rounds. Also, whereas in small scale tests the improvement in performance between the 5^{th} and 10^{th} rounds seemed rather small, it is more pronounced in the larger scale tests. This is logical, as the total number of replicas for any given key is greater for larger values of n. Thus, R^5N is able to find new replicas in each additional round of PUT requests, thus increasing the chance of GET requests succeeding.

For the 60,000 peer test in Figure 7.19b Kademlia finds data successfully less than 10% of the time without any malicious peers, while R^5N achieves above 80% after 10 PUT operations. For the 80,000 peer test in Figure 7.19c Kademlia finds data successfully less than 5% of the time without any malicious peers, while R^5N improves to around 50% after enough PUT requests are performed. Of course, we expect these numbers to improve even more with a greater number of total connections for even the 80,000 peer network. Here we may see the limit to which R^5N can achieve high rates of success in networks with very few connections. However, the performance improvement over R-Kademlia is obvious, and any restricted route networks would certainly benefit from a 50 percent success rate with as opposed to a less than 5 percent success rate.

7.8 Conclusion

The presented algorithm needs to first create a sufficient number of replicas. A network with n nodes of degree c is expected to have $\frac{n}{c+1}$ nearest peers. According to the birthday paradox, $\sqrt{\frac{n}{c+1}}$ replicas would need to be created in order for a GET request to succeed with about 50% probability (at this probability, a small constant number of repetitions can be used to get acceptable overall success rates). As we have shown experimentally, the relationship between the number of replicas in the network and the number of hops required for the respective PUT operations is almost linear (Figure 7.7).

Since GET requests have complexity $O(\log n)$ (Section 7.3.3), routing in a Small-World network with R^5N scales with $O(\sqrt{n} \cdot \log n)$. Note that this does not hold in sparse graphs with large diameter or graphs that are not expander graphs, such as a circle, because the routing table could not be sufficiently populated even using distance-vector augmentation (Chapter 5) with a small, constant maximum path length.

While the randomized nature of R^5N helps to circumvent Eclipse attacks, a large number of adversarial nodes can still have a significant performance impact. R^5N is robust against a range of well-known attacks on

DHTs, including poisoning attacks, Sybil attacks and Eclipse attacks. Its performance is comparable to that of a recursive implementation of Kademlia even in unrestricted network topologies which Kademlia was designed for. R^5N still performs well in restricted-route topologies, including Small-World networks. In contrast, Kademlia fails to locate data in Small-World networks in most cases.

8. CONCLUSION AND FUTURE WORK

Our research on routing in open P2P networks has resulted in a number of significant contributions. Based on our analysis, the developers of Freenet made modifications to their routing algorithm to minimize the impact of our location swapping attack. Additionally, we explained key clustering in the network due to natural churn processes. This further helped the Freenet developers understand some unexpected characteristics of the deployed network. Similarly, our analysis of the Tor network revealed both that a previous de-anonymizing attack was no longer viable, and that a protocol flaw existed that allowed a asymmetric denial-of-service attack. This flaw allowed a modified version of the original de-anonymizing attack to succeed. As a result of this research, we created a patch fixing the protocol flaw which became part of the Tor implementation.

The emulation framework which we have created disproves the notion that large scale P2P security evaluations can only be done by running simulations. This should both help researchers in performing more realistic evaluations, and reduce the amount of work required due to maintenance of two separate implementations.

Current DHTs are unable to operate securely in restricted-route networks. R^5N provides an efficient, secure DHT routing algorithm which operates in many different underlay topologies. It is our expectation that R^5N will be used as a building block for allowing decentralized services to operate in these network topologies, where previous solutions do not operate properly.

The usefulness of randomization for mitigating malicious and mis-configured peers in a restricted route network justifies the small drop in efficiency.

8.1 Future Work

Replication is undeniably useful in DHTs for providing reliability and fault-tolerance. Storing data at a single peer in a network is simply unacceptable when peers may go off-line at any time, and generally can not be trusted. While our current design does not support replication in unrestricted topologies, in future work we plan to investigate randomizing keys [140] in addition or instead of randomizing routing. Randomizing keys causes data to be stored at multiple nodes, even in fully connected networks. For this reason, key randomization works best to provide replication and defense against routing attacks in unrestricted networks.

However, there are two key problems with implementing key randomization in R^5N. First, key randomization should only be used if the topology is in fact unrestricted. If key randomization *and* routing randomization is used in a restricted-route network, the probability of successfully locating data could drop significantly. We speculate that there is some combination of both route and key randomization that works best for certain topologies.

Additional future research directions involve implementing other P2P algorithms in the GNUnet framework, and performing further comparisons between R^5N and these. A feature of the GNUnet framework is that any algorithm can be used, possibly even multiple algorithms concurrently. We believe that our implementation of the GNUnet testing/emulation framework should remain scalable up to millions of peers. However, we have thus far lacked the hardware resources to attempt to run such tests. Therefore, further testing our framework on larger clusters remains a goal for future work. Finally, our implementation is not intended to be used only in the laboratory, and the next step of action will be to actually deploy the implementation in the real world, and get feedback from users on usability, bugs, etc.

BIBLIOGRAPHY

[1] J. P. Ahulló and P. G. López. Planetsim: An extensible simulation tool for peer-to-peer networks and services. In *Proceedings of the 9th International Conference on Peer-to-Peer Computing*, pages 85–86, September 2009.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] J. Appelbaum. Detecting certificate authority compromises and web browser collusion. https://blog.torproject.org/blog/detecting-certificate-authority-compromises-and-web-browser-collusion, March 2011. The Tor Blog.

[4] C. Avin and G. Ercal. On the cover time and mixing time of random geometric graphs. *Theoretical Computer Science*, 380:2–22, July 2007.

[5] Y. Azar, A. Z. Broder, A. R. Karlin, N. Linial, and S. Phillips. Biased random walks. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, STOC '92, pages 1–9. ACM, 1992.

[6] A. Back, U. Möller, and A. Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Proceedings of Information Hiding Workshop*, pages 245–257. Springer-Verlag, LNCS 2137, April 2001.

[7] E. Bangeman. Study: Bittorrent sees big growth, limewire still #1 p2p app. http://arstechnica.com/old/content/2008/04/study-bittorren-sees-big-growth-limewire-still-1-p2p-app.ars, September 2007.

[8] I. Baumgart, B. Heep, and S. Krause. Oversim: A flexible overlay network simulation framework. In *Proceedings of 10th IEEE Global Internet Symposium*, pages 79–84, May 2007.

[9] N. Beijar. Zone Routing Protocol (zrp). http://www.netlab.hut.fi/opetus/s38030/k02/Papers/08-Nicklas.pdf, April 2002. Licentiate course on Telecommunications Technology.

[10] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[11] S. Bellovin. Using the domain name system for system break-ins. In *Proceedings of the 5th Conference on USENIX UNIX Security Symposium*, volume 5, pages 18–18, Berkeley, CA, USA, June 1995. USENIX Association.

[12] S. M. Bellovin. Security problems in the tcp/ip protocol suite. *SIGCOMM Computer Communication Review*, 19:32–48, April 1989.

[13] S. M. Bellovin. Security aspects of napster and gnutella. http://www.research.att.com/smb/talks/, August 2000. Invited Talk.

[14] S. M. Bellovin. A look back at "security problems in the tcp/ip protocol suite". In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 229–249, Washington, DC, USA, 2004. IEEE Computer Society.

[15] I. Benjamini, G. Kozma, and N. Wormald. The mixing time of the giant component of a random graph. October 2006. http://arxiv.org/abs/math/0610459v1.

[16] K. Bennett and C. Grothoff. gap: Practical anonymous networking. In *Proceedings of the 3rd International Workshop on Privacy Enhancing Technologies*, ser-LNCS, pages 141–160. Springer-Verlag, 2003.

[17] K. Bennett, C. Grothoff, T. Horozov, and J. T. Lindgren. An encoding for censorship-resistant sharing. Technical report, 2003.

[18] M. Bertier, F. Bonnet, A.-M. Kermarrec, V. Leroy, S. Peri, and M. Raynal. D2ht: The best of both worlds, integrating rps and dht. In *Proceedings of the 2010 European Dependable Computing Conference*, EDCC '10, pages 135–144, Washington, DC, USA, 2010. IEEE Computer Society.

[19] bgpmon. Chinese isp hijacked 10% of the internet. http://bgpmon.net/blog/?p=282, April 2010. BGPmon.net Blog.

[20] U. Black. *IP routing protocols: RIP, OSPF, BGP, PNNI and Cisco Routing Protocols*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[21] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, July 1970.

[22] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz. Denial of service or denial of security? how attacks on reliability can compromise anonymity. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 92–102, New York, NY, USA, October 2007. ACM.

[23] D. Bruschi, A. Orgnaghi, and E. Rosti. S-arp: a secure address resolution protocol. In *Proceedings of the 19th Annual Computer Security Applications Conference*, ACSAC '03, pages 66–74, Washington, DC, USA, December 2003. IEEE Computer Society.

[24] J. Calvet, C. R. Davis, J. M. Fernandez, W. Guizani, M. Kaczmarek, J.-Y. Marion, and P.-L. St-Onge. Isolated virtualised clusters: Testbeds for high-risk security experimentation and training. In *Proceedings of the 3rd International Conference on Cyber Security Experimentation and Test*, CSET'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[25] J. Calvet, C. R. Davis, J. M. Fernandez, J.-Y. Marion, P.-L. St-Onge, W. Guizani, P.-M. Bureau, and A. Somayaji. The case for in-the-lab botnet experimentation: Creating and taking down a 3000-node botnet. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 141–150. ACM, 2010.

[26] B. Carpenter. Architectural principles of the internet. RFC 1958, IETF, http://www.ietf.org/rfc/rfc1958.txt, June 1996. Updated by RFC 3439.

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

[28] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, February 1981.

[29] R. Chertov, S. Fahmy, and N. B. Shroff. Fidelity of network simulation and emulation: A case study of tcp-targeted denial of service attacks. *ACM Transactions on Modeling and Computer Simulation*, 19(1):4:1–4:29, December 2008.

[30] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, pages 46–66. Springer-Verlag New York, Inc., 2001.

[31] CNN. Pakistan move knocked out youtube. http://www.cnn.com/2008/WORLD/asiapcf/02/25/pakistan.youtube/index.html, February 2008.

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

[33] 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, pages 202–215. ACM, 2001.

[34] W. Dai. Two attacks against freedom. http://www.weidai.com/freedom-attacks.txt, 2000.

[35] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 2–15, May 2003.

[36] G. Danezis, C. Lesniewski-Laas, M. F. Kaashoek, and R. Anderson. Sybil-resistant dht routing. In *Proceedings of the 10th European Symposium on Research in Computer Security*, ESORICS '05, pages 305–318. Springer-Verlag, 2005.

[37] e. a. Daniel Stenberg. libcurl. http://curl.haxx.se/libcurl/, 1998–2009. Open Source C-based multi-platform file transfer library.

[38] M. Dell'Amico and Y. Roudier. A measurement of mixing time in social networks. In *Proceedings of the 5th International Workshop on Security and Trust Management*, STM '09, September 2009.

[39] Y. Desmedt and K. Kurosawa. How to break a practical mix and design a new one. In *Proceedings of the 19th international Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'00, pages 557–572, Berlin, Heidelberg, 2000. Springer-Verlag.

[40] C. Diaz and A. Serjantov. Generalising mixes. In *Proceedings of Privacy Enhancing Technologies Workshop*, PET '03, pages 18–31. Springer-Verlag, LNCS 2760, March 2003.

[41] T. Dierks and C. Allen. The tls protocol version 1.0. RFC 2246, IETF, http://www.ietf.org/rfc/rfc2246.txt, January 1999.

[42] R. Dingledine. Tor proposal 110: Avoiding infinite length circuits. https://svn.torproject.org/svn/tor/trunk/doc/spec/proposals/110-avoid-infinite-circuits.txt, March 2007.

[43] R. Dingledine. Tor bridges specification. Technical report, The Tor Project, https://svn.torproject.org/svn/tor/trunk/doc/spec/bridges-spec.txt, 2008.

[44] R. Dingledine and N. Mathewson. Design of a blocking-resistant anonymity system. Technical report, The Tor Project, https://svn.torproject.org/svn/projects/design-paper/blocking.html, November 2006.

[45] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, August 2004.

[46] T. T. A. Dinh, G. Theodoropoulos, and R. Minson. Evaluating large scale distributed simulation of p2p networks. In *Proceedings of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 51–58. IEEE Computer Society, 2008.

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

[48] B. Drain. Eve evolved: Eve online's server model. http://www.massively.com/2008/09/28/eve-evolved-eve-onlines-server-model/, September 2008.

[49] C. DSS. Gnutella protocol specification v0.4. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf, 2001.

[50] D. Eastlake. Domain name system security extensions. RFC 2535, IETF, http://www.ietf.org/rfc/rfc2535.txt, March 1999. Obsoleted by RFCs 4033, 4034, 4035, updated by RFCs 2931, 3007, 3008, 3090, 3226, 3445, 3597, 3655, 3658, 3755, 3757, 3845.

[51] P. Eckersley and J. Burns. An observatory for the ssliverse. https://www.eff.org/files/DefconSSLiverse.pdf, July 2010. Presented at Defcon 18.

[52] J. Edwards. *Peer-to-Peer Programming on Groove*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[53] K. Egevang and P. Francis. The ip network address translator (nat). RFC 1631, IETF, http://tools.ietf.org/html/rfc1631, May 1994.

[54] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631 (Informational), http://www.ietf.org/rfc/rfc1631.txt, May 1994. Obsoleted by RFC 3022.

[55] M. W. et al. Mariadb. http://mariadb.org, May 2011.

[56] N. Evans and C. Grothoff. R5n: Randomized recursive routing for restricted-route networks. In *5th International Conference on Network and System Security*, Milan, Italy, 2011. IEEE.

[57] N. S. Evans. Routing in the dark: Pitch black. Master's thesis, University of Denver, May 2009.

[58] N. S. Evans, R. Dingledine, and C. Grothoff. A practical congestion attack on tor using long paths. In *Proceedings of the 18th USENIX Security Symposium*, pages 33–50. USENIX, 2009.

[59] N. S. Evans, C. GauthierDickey, and C. Grothoff. Routing in the dark: Pitch black. In *Proceedings of the Annul Computer Security Conference*, ACSAC '07, pages 305–314. IEEE Computer Society, 2007.

[60] N. S. Evans and C. Grothoff. Beyond simulation: Large-scale distributed emulation of p2p protocols. In *Proceedings of the 3rd international conference on Cyber security experimentation and test*, CSET'11, 2011.

[61] C. E. A. Falk. An update on the geni project. *SIGCOMM Computer Communication Review*, 39:28–34, June 2009.

[62] S. Fanning. Napster. http://www.napster.com/, 1999.

[63] R. A. Ferreira, C. Grothoff, and P. Ruth. A transport layer abstraction for peer-to-peer networks. In *Proceedings of the 4th International Workshop on GRID Computing*, pages 398–403. IEEE Computer Society, November 2003.

[64] R. A. Ferreira, M. K. Ramanathan, A. Awan, A. Grama, and S. Jagannathan. Search with probabilistic guarantees in unstructured peer-to-peer networks. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, pages 165–172, Washington, DC, USA, 2005. IEEE Computer Society.

[65] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, IETF, http://www.ietf.org/rfc/rfc2616.txt, June 1999.

[66] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[67] N. Fountoulakis and B. A. Reed. The evolution of the mixing rate of a simple random walk on the giant component of a random graph. *Random Structures and Algorithms*, 33:68–86, August 2008.

[68] M. J. Freedman and R. Morris. Tarzan: a peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 193–206, New York, NY, USA, November 2002. ACM.

[69] M. J. Freedman, E. Sit, J. Cates, and R. Morris. Introducing tarzan, a peer-to-peer anonymizing network layer. In *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 121–129, London, UK, 2002. Springer-Verlag.

[70] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1 of *INFOCOM '04*, pages 1–11, March 2004.

[71] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding routing information. In *Proceedings of Information Hiding: First International Workshop*, pages 137–150. Springer-Verlag, LNCS 1174, May 1996.

[72] M. T. Goodrich, M. J. Nelson, and J. Z. Sun. The rainbow skip graph: A fault-tolerant constant-degree distributed data structure. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm*, pages 384–393. ACM Press, 2006.

[73] C. Gülcü and G. Tsudik. Mixing e-mail with babel. In *Proceedings of the Network and Distributed Security Symposium*, NDSS '96, pages 2–16. IEEE, February 1996.

[74] J. Han and Y. Liu. Rumor riding: Anonymizing unstructured peer-to-peer systems. In *Proceedings of the 2006 IEEE International Conference on Network Protocols*, ICNP '06, pages 22–31, Washington, DC, USA, November 2006. IEEE Computer Society.

[75] J. Hautakorpi and G. Camarillo. Evaluation of dhts from the viewpoint of interpersonal communications. In *Proceedings of the 6th International Conference on Mobile and Ubiquitous Multimedia*, MUM '07, pages 74–83. ACM, 2007.

[76] Q. He, M. H. Ammar, G. F. Riley, H. Raj, and R. Fujimoto. Mapping peer behavior to packet-level details: A framework for packet-level simulation of peer-to-peer systems. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, pages 71–78, October 2003.

[77] L. T. Heberlein and M. Bishop. Attack class: Address spoofing. In *Proceedings of the 19th National Information Systems Security Conference*, pages 371–377, 1996.

[78] C. Hedrick. Routing information protocol. RFC 1058 (Historic), IETF, http://www.ietf.org/rfc/rfc1058.txt, June 1988. Updated by RFCs 1388, 1723.

[79] B. Heep. R/kademlia: Recursive and topology-aware overlay routing. In *Proceedings of the Conference on Telecommunication Networks and Applications*, pages 102–107, November 2010.

[80] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In

Proceedings of the USENIX 2008 Annual Technical Conference, pages 113–128, Berkeley, CA, USA, 2008. USENIX Association.

[81] D. Hildebrandt, L. Bischofs, and W. Hasselbring. Realpeer–a framework for simulation-based development of peer-to-peer systems. In *Proceedings of the Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pages 490–497, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[82] N. Hopper, E. Y. Vasserman, and E. Chan-Tin. How much anonymity does network latency leak? In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 82–91, New York, NY, USA, October 2007. ACM.

[83] Y.-C. Hu, A. Perrig, and M. Sirbu. Spv: secure path vector routing for securing bgp. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 179–192. ACM Press, 2004.

[84] C. Hurley, S. Chen, and J. Karim. YouTube: broadcast yourself. http://www.youtube.com, 2009.

[85] G. Huston. Ipv4 address report. http://www.potaroo.net/tools/ipv4/, March 2010.

[86] P. Iyer and U. Warrier. Internetgatewaydevice:1 device template version 1.01. http://www.upnp.org/specs/gw/UPnP-gw-InternetGatewayDevice-v1-Device.pdf, November 2001.

[87] J. Kangasharju, U. Schmidt, D. Bradler, and J. Schröder-Bernhardi. Chunksim: Simulating peer-to-peer content distribution. In *Proceedings of the 2007 Spring Simulaiton Multiconference*, volume 1, pages 25–32, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[88] A. Kapela and A. Pilosov. Stealing the internet - a routed, wide-area, man in the middle attack. http://defcon.org/images/defcon-16/dc16-presentations/defcon-16-pilosov-kapela.pdf, 2008. Presented at DEFCON 16.

[89] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, December 1998.

[90] F. Keil, D. Schmidt, and et al. Privoxy - a privacy enhancing web proxy. http://www.privoxy.org/, August 2002.

[91] D. Kesdogan, J. Egner, and R. Büschkes. Stop-and-go mixes: Providing probabilistic anonymity in an open system. In *Proceedings of the 2nd International Workshop on Information Hiding*, pages 83–98, London, UK, 1998. Springer-Verlag, LNCS 1525.

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

[93] J. M. Kleinberg. The small-world phenomenon: An algorithm perspective. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, STOC '00, pages 163–170. ACM Press, 2000.

[94] N. Kotilainen, M. Vapa, T. Keltanen, A. Auvinen, and J. Vuori. P2prealm - peer-to-peer network simulator. In *Proceedings of the 11th International Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks*, pages 93–99, 2006.

[95] M. Kozlovszky, A. Balasko, and A. Varga. Enabling omnet++-based simulations on grid systems. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 67:1–67:7, Brussels, Belgium, 2009.

Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST).

[96] S. Krco, D. Cleary, and D. Parker. P2p mobile sensor networks. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, HICSS '05, pages 324c–324c, Washington, DC, USA, 2005. IEEE Computer Society.

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

[98] O. Landsiedel, K. A. Lehmann, and K. Wehrle. T-dht: Topology-based distributed hash tables. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, pages 143–144, 2005.

[99] O. Landsiedel, A. Pimenidis, K. Wehrle, H. Niedermayer, and G. Carle. Dynamic multipath onion routing in anonymous peer-to-peer overlay networks. In *Proceedings of the Global Telecommunications Conference*, IEEE GLOBECOM '07, pages 64–69, November 2007.

[100] D. A. Levin, Y. Peres, and E. L. Wilmer. *Markov Chains and Mixing Times.* American Mathematical Society, 2006.

[101] B. N. Levine, M. K. Reiter, C. Wang, and M. K. Wright. Timing attacks in low-latency mix-based systems. In *Proceedings of Financial Cryptography*, pages 251–265. Springer-Verlag, LNCS 3110, February 2004.

[102] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, IPTPS '04, pages 87–99, February 2004.

[103] T. Locher, D. Mysicka, S. Schmid, and R. Wattenhofer. Poisoning the kad network. In *Proceedings of the 11th International Conference on Distributed Computing and Networking*, ICDCN '10, pages 195–206, January 2010.

[104] L. Lov'asz. Random walks on graphs: A survey. *Combinatorics*, 2(1):1–46, 1993.

[105] L. Ltd. Interactive, live tv. http://www.livestation.com, 2008.

[106] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international Conference on Supercomputing*, ICS '02, pages 84–95. ACM, 2002.

[107] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, volume 4 of *USITS'03*, pages 10–10, Berkeley, CA, USA, 2003. USENIX Association.

[108] G. S. Manku, M. Naor, and U. Wieder. Know thy neighbor's neighbor: the power of lookahead in randomized p2p networks. In *Proceedings of the 36th ACM Symposium on Theory of Computing*, pages 54–63, 2004.

[109] M. Marlinspike. Ssl and the future of authenticity. http://blog.thoughtcrime.org/ssl-and-the-future-of-authenticity, April 2011. Thoughtcrime Labs blog.

[110] O. C. Martin and P. Šulc. Return probabilities and hitting times of random walks on sparse erdös-rényi graphs. *Phys. Rev. E*, 81(3):031111, Mar 2010.

[111] I. Martinez-Yelmo, R. Cuevas, C. Guerrero, and A. Mauthe. Routing performance in a hierarchical dht-based overlay network. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '08, pages 508–515, Washington, DC, USA, 2008. IEEE Computer Society.

[112] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, 2002. Springer-Verlag.

[113] J. Mclachlan and N. Hopper. Don't clog the queue! circuit clogging and mitigation in p2p anonymity schemes. In *Financial Cryptography and Data Security*, pages 31–46, Berlin, Heidelberg, 2008. Springer-Verlag.

[114] E. L. Merrer, A. M. Kermarrec, and L. Massoulie. Peer-to-peer size estimation in large and dynamic networks: A comparative study. In *Proceedngs of the 15th IEEE International Symposium on High Performance Distributed Computing*, pages 7–17, 2006.

[115] S. Milgram. The small-world problem. *Psychology Today*, 2:60–67, 1967.

[116] J. Mirkovic, T. V. Benzel, T. Faber, R. Braden, J. Wroclawski, and S. Schwab. The deter project: Advancing the science of cyber security experimentation and test. In *Proceedings of the 2010 IEEE International Conference on Technologies for Homeland Security*, pages 1–7, November 2010.

[117] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach. Ap3: Cooperative, decentralized anonymous communication. In *Proceedings of the 11th ACM SIGOPS European Workshop*, SIGOPS-EW '04, page 30. ACM, September 2004.

[118] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), IETF, http://www.ietf.org/rfc/rfc1035.txt, November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343.

[119] U. Möller, L. Cottrell, P. Palfrader, and L. Sassaman. Mixmaster protocol — version 2. IETF Internet Draft, http://tools.ietf.org/html/draft-sassaman-mixmaster-03, December 2004.

[120] A. Montresor and M. Jelasity. Peersim: A scalable p2p simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer Computing*, pages 99–100, September 2009.

[121] J. Moy and et. al. The ospf specification. RFC 1131, IETF, http://tools.ietf.org/html/rfc1131, October 1989. Obsoleted by RFC-1247.

[122] A. Muller, N. Evans, C. Grothoff, and S. Kamkar. Autonomous nat traversal. In *Proceedings of the 10th IEEE International Conference on Peer-to-Peer Computing*, IEEE P2P '10, pages 61–64, Delft, The Netherlands, August 2010. IEEE.

[123] A. Muller, A. Klenk, and G. Carle. Behavior and classification of nat devices and implications for nat-traversal. *Network, IEEE*, 22:14–19, September 2008.

[124] A. Müller, A. Klenk, and G. Carle. On the applicability of knowledge based nat-traversal for home networks. In *Proceedings of the 7th International IFIP-TC6 Networking Conference on AdHoc and Sensor Networks, Wireless Networks, Next Generation Internet*, NETWORKING'08, pages 264–275, Berlin, Heidelberg, May 2008. Springer-Verlag.

[125] S. J. Murdoch. *Covert Channel Vulnerabilities in Anonymity Systems.* PhD thesis, December 2007.

[126] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of tor. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 183–195, Washington, DC, USA, May 2005. IEEE Computer Society.

[127] M. Nafaa and N. Agoulmine. Analysing joost peer-to-peer iptv protocol. In *Proceedings of the 11th IFIP/IEEE International Symposium on Integrated Network Management*, IM'09, pages 291–294, Piscataway, NJ, USA, 2009. IEEE Press.

[128] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *SIGCOMM Computer Communication Review*, 37:95–98, March 2007.

[129] A. Nambiar and M. Wright. Salsa: A structured approach to large-scale anonymity. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 17–26, New York, NY, USA, October 2006. ACM.

[130] Z. Ou, E. Harjula, O. Kassinen, and M. Ylianttila. Performance evaluation of a kademlia-based communication-oriented p2p system under churn. *Computer Networks*, 54:689–705, April 2010.

[131] L. Øverlier and P. Syverson. Locating hidden servers. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 100–114, Washington, DC, USA, May 2006. IEEE Computer Society.

[132] J. Palacios. Bounds on expected hitting times for a random walk on a connected graph. *Linear Algebra and its Applications*, 141:241 – 252, 1990.

[133] V. Pappas, E. Athanasopoulos, S. Ioannidis, and E. P. Markatos. Compromising anonymity using packet spinning. In *Proceedings of the 11th Information Security Conference*, ISC '08, pages 161–174, Berlin, Heidelberg, September 2008. Springer-Verlag.

[134] V. Pappas, D. Massey, A. Terzis, and L. Zhang. A comparative study of the dns design with dht-based alternatives. In *Proceedings of the 25th IEEE International Conference on Computer Communications*, pages 1–13, April 2006.

[135] P. Parkes. Skype downtime today. http://blogs.skype.com/en/2010/12/skype_downtime_today.html, December 2010. Skype - The Big Blog.

[136] G. Pei, M. Gerla, and T. wei Chen. Fisheye state routing in mobile ad hoc networks. In *Proceedings of the ICDCS Workshop on Wireless Networks and Mobile Computing*, volume 1, pages 71–74, June 2000.

[137] M. Perry and S. Squires. Torbutton. https://www.torproject.org/torbutton/, 2009.

[138] A. Pfitzmann, B. Pfitzmann, and M. Waidner. Isdn-mixes: Untraceable communication with very small bandwidth overhead. In *Proceedings of the GI/ITG Conference on Communication in Distributed Systems*, pages 451–463, February 1991.

[139] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 311–320. ACM, 1997.

[140] B. Polot. Adapting blackhat approaches to enhance the resillience of whitehat application scenarios. Master's thesis, 2010.

[141] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Proceedings of the Workshop on Algorithms and Data Structures*, WADS '89, pages 437–449, London, UK, 1989. Springer-Verlag.

[142] G. N. Purdy. *Linux iptables Pocket Reference.* O'Reilly Media, August 2004.

[143] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 31 of *SIGCOMM '01*, pages 161–172. ACM, August 2001.

[144] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for dhts: Some open questions. In *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 45–52, London, UK, 2002. Springer-Verlag.

[145] Y. Rehkhter and T. Li. A border gateway protocol 4 (bgp-4). RFC 1771, IETF, http://www.ietf.org/rfc/rfc1771.txt, 1995.

[146] M. Rennhard and B. Plattner. Introducing morphmix: Peer-to-peer based anonymous internet usage with collusion detection. In *Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society*, WPES '02, pages 91–102, New York, NY, USA, November 2002. ACM.

[147] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 127–140, Berkeley, CA, USA, June 2004. USENIX Association.

[148] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: A public dht service and its uses. *SIGCOMM Computer Communication Review*, 35:73–84, 2005.

[149] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, volume 1, pages 267–280, Berkeley, CA, USA, 2004. USENIX Association.

[150] J. Rose, C. Hall, and A. Carzaniga. Spinneret: A log random substrate for p2p networks. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[151] J. Rosenberg and R. M. et. al. Session traversal utilities for nat (stun). RFC 5389, IETF, http://tools.ietf.org/html/rfc5389, October 2008.

[152] J. Rosenberg, R. Mahy, and P. Matthews. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). RFC 5766 (Review Copy), IETF, http://tools.ietf.org/html/rfc5766, April 2010.

[153] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, Middleware '01, pages 329–350, London, UK, 2001. Springer-Verlag.

[154] A. I. T. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 188–201. ACM, October 2001.

[155] V. Sacramento. Vulnerability in the sending requests control of bind versions 4 and 8 allows dns spoofing. http://www.rnp.br/cais/alertas/2002/cais-ALR-19112002a.html, November 2002.

[156] P. Saint-Andre. Streaming xml with jabber/xmpp. *IEEE Internet Computing*, 9(5):82–89, 2005.

[157] P. Salihundam, S. Jain, T. Jacob, S. Kumar, V. Erraguntla, Y. Hoskote, S. Vangal, G. Ruhl, and N. Borkar. A 2 tb/s 6 x 4 mesh network for a single-chip cloud computer with dvfs in 45 nm cmos. *IEEE Journal of Solid-State Circuits*, 46(4):757–766, April 2011.

[158] O. Sandberg. *Searching in a Small World.* PhD thesis, 2005.

[159] O. Sandberg. Distributed routing in small-world networks. In *Algorithm Engineering and Experiments*, pages 179–188. SIAM, 2006.

[160] D. Sax. Dns spoofing malicious cache poisoning. https://www.giac.org/paper/gsec/189/dns-spoofing-malicious-cache-poisoning/100664, 2000. SANS Institute.

[161] M. Schlosser, T. Condie, and S. Kamvar. Simulating a file-sharing p2p network. Technical Report 2003-28, Stanford InfoLab, 2003.

[162] H. Schulze and K. Mochalski. Internet study 2007. http://www.ipoque.com/resources/internet-studies/internet-study-2007, 2007. ipoque.

[163] A. Serjantov, R. Dingledine, and P. Syverson. From a trickle to a flood: Active attacks on several mix types. In *Revised Papers from the 5th International Workshop on Information Hiding*, pages 36–52, London, UK, 2002. Springer-Verlag, LNCS 2578.

[164] V. Shmatikov and M.-H. Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *Proceedings of the 11th European Symposium on Research in Computer Security*, ESORICS '06, pages 236–252, September 2006.

[165] K. Shudo, Y. Tanaka, and S. Sekiguchi. Overlay weaver: An overlay construction toolkit. *Computer Communications*, 31(2):402–412, 2008. Special Issue: Foundation of Peer-to-Peer Computing.

[166] S. Sioutas, G. Papaloukopoulos, E. Sakkopoulos, K. Tsichlas, and Y. Manolopoulos. A novel distributed p2p simulator architecture: D-p2p-sim. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 2069–2070. ACM, 2009.

[167] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 261–269, London, UK, 2002. Springer-Verlag.

[168] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), http://www.ietf.org/rfc/rfc3022.txt, Jan. 2001.

[169] P. Srisuresh, B. Ford, and D. Kegel. State of peer-to-peer (p2p) communication across network address translators (nats). RFC 5128, IETF, http://tools.ietf.org/html/rfc5128, March 2008.

[170] M. Steiner, T. En-Najjary, and E. W. Biersack. Exploiting kad: Possible uses and misuses. *SIGCOMM Computer Communication Review*, 37(5):65–70, October 2007.

[171] M. Steiner, T. En-Najjary, and E. W. Biersack. A global view of kad. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, pages 117–122. ACM, 2007.

[172] J. Stewart. Dns cache poisoning - the next generation. http://www.secureworks.com/research/articles/cachepoisoning, 2002. Dell Secureworks Research Labs.

[173] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160. ACM, 2001.

[174] L. Student Member-Ramaswamy, B. Student Member-Gedik, and L. Member-Liu. A distributed approach to node clustering in decentralized peer-to-peer networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(9):814–829, 2005.

[175] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, volume 13 of *IMC '06*, pages 189–202. ACM Press, 2006.

[176] P. Syverson, D. Goldschlag, and M. Reed. Anonymous connections and onion routing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 44–54, Oakland, California, 1997.

[177] P. Syverson, G. Tsudik, M. Reed, and C. Landwehr. Towards an analysis of onion routing security. In *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, pages 96–114. Springer-Verlag, LNCS 2009, July 2000.

[178] W. W. Terpstra, J. Kangasharju, C. Leng, and A. P. Buchmann. Bubblestorm: Resilient, probabilistic and exhaustive. In *Proceedngs of the 2007 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 49–60. ACM, 2007.

[179] G. Urdaneta, G. Pierre, and M. V. Steen. A survey of dht security techniques. *ACM Computer Survey*, 43:8:1–8:49, February 2011.

[180] A. Varga. The omnet++ discrete event simulation system. *In Proceedings of the European Simulation Multiconference*, pages 319–324, June 2001.

[181] K. V. Vishwanath, D. Gupta, A. Vahdat, and K. Yocum. Modelnet: Towards a datacenter emulation environment. In *Proceedings of the 9th IEEE International Conference on Peer-to-Peer Computing*, pages 81–82, September 2009.

[182] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks*, pages 37–42. ACM, August 2009.

[183] D. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[184] R. Wiangsripanawan, W. Susilo, and R. Safavi-Naini. Design principles for low latency anonymous network systems secure against timing attacks. In *Proceedings of the 5th Australasian Symposium on ACSW Frontiers*, ACSW '07, pages 183–191, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[185] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybilguard: Defending against sybil attacks via social networks. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 267–278, New York ,NY, USA, 2006. ACM Press.

[186] N. Zennström and J. Friis. Skype. http://www.skype.com, 2003.

[187] 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, 22(1):41–53, January 2004. Special Issue on Service Overlay Networks, to appear.

[188] L. Zhuang, F. Zhou, U. C. Berkeley, B. Y. Zhao, and A. Rowstron. Cashmere: Resilient anonymous routing. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*. ACM, May 2005.

[189] H. Zhuge, X. Chen, X. Sun, and E. Yao. Hring: A structured p2p overlay based on harmonic series. *IEEE Transactions on Parallel and Distributed Systems*, 19:145–158, February 2008.

[190] zzz, postman, and et. al. I2p. http://www.i2p2.de, 2003–2011.

ISBN 3-937201-26-2

ISSN 1868-2634 (print)
ISSN 1868-2642 (electronic)